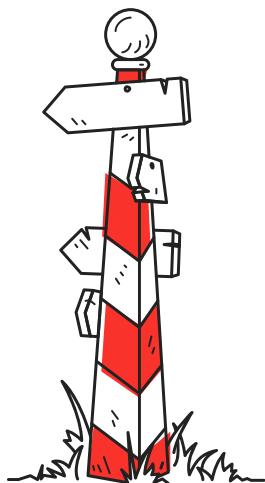


2024

ВСЁ ЧТО НУЖНО ДЛЯ ДОСТИЖЕНИЯ ЦЕЛИ

# Одна платформа, множество путей.

«Из коробки» Laravel предлагает элегантные решения для множества функций, необходимых всем современным приложениям.



Inertia



Livewire



API

## Livewire

Современный способ создания динамических интерфейсов с использованием серверных шаблонов вместо JavaScript-фреймворков. Он сочетает в себе простоту и быстроту разработки серверного приложения с пользовательским опытом JavaScript SPA (Single Page Application). Вам нужно увидеть это, чтобы поверить.

```

use Livewire\Component;

class Search extends Component
{
    public $search = '';

    public function render()
    {
        $users = User::search($this->search)->get();

        return view('livewire.search', [
            'users' => $users,
        ]);
    }
}

<div>
    <input wire:model="search"
        type="text"
        placeholder="Search users..." />

    <ul>
        @foreach ($users as $user)
            <li>{{ $user->username }}</li>
        @endforeach
    </ul>
</div>

```

При использовании Livewire вам не нужен JavaScript для управления DOM или состоянием - вы просто добавите его для некоторых продуманных взаимодействий. Alpine.js - идеальная легковесная JavaScript-библиотека для сочетания с вашим приложением на Livewire.

По мере изменения состояния вашего компонента Livewire, ваш фронтенд автоматически будет обновляться. Но Livewire не останавливается на этом. Поддержка реального времени для проверки данных, обработки событий, загрузки файлов, авторизации и многого другого включена.

**Как это работает?**

Livewire отрисовывает ваш HTML на сервере с использованием языка шаблонов Blade. Он автоматически добавляет необходимый JavaScript, чтобы страница стала реактивной, а также автоматически перерисовывает компоненты и обновляет DOM при изменении данных.

## Погрузитесь прямо со старта.

Независимо от того, предпочитаете ли вы Livewire или React, стартовые наборы Laravel позволят вам сразу же приступить к делу. За считанные минуты вы можете получить полнофункциональное приложение, сочетающее Laravel и Tailwind с выбранным вами интерфейсом.

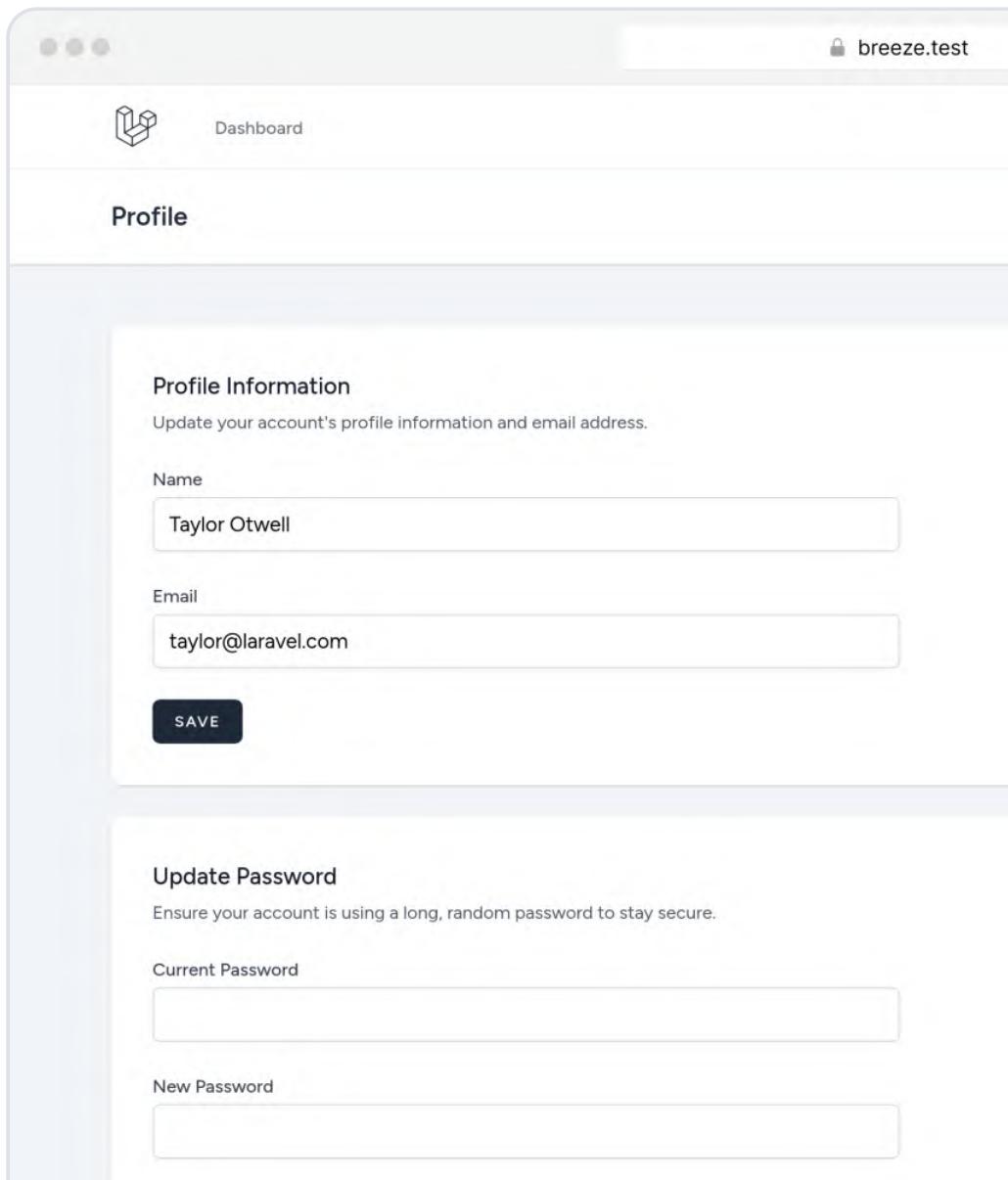


### Laravel Breeze

Легкий стартовый комплект, который включает в себя шаблоны управления профилями пользователей для аутентификации в стиле Tailwind.

- Регистрация пользователя и вход в систему
- Сброс пароля
- Подтверждение адреса электронной почты
- Управление профилями пользователей
- Blade или Inertia (с Vue или React)
- Дополнительная поддержка TypeScript

- Дополнительная поддержка темного режима



Настраивать окружение для новичка может быть непростой задачей. Однако, есть несколько простых и удобных способов быстро и легко запустить Laravel и сосредоточиться на разработке приложения.



Laravel для Mac

Laravel Valet предоставляет простой и минималистичный способ настройки вашей среды разработки для запуска приложений, а также обеспечивает доступ к ним через `*.test` домен.



## Laravel для Docker

Если вам нужна гибкость и изоляция, Laravel Sail предоставляет легкий интерфейс командной строки для работы с Docker. Даже если у вас нет опыта работы с Docker.



## Маршрутизация

Маршрутизация (Routing) позволяет определить, как приложение должно отвечать на разные URL-адреса. Это позволяет легко настраивать маршруты для обработки запросов и определять, какие действия и контроллеры должны быть вызваны при поступлении запроса.

[Подробнее →](#)



## Шаблоны Blade

Вставляйте переменные, используйте условия, циклы и другие операции в шаблонах, что делает их более читабельными и удобными для разработки.

[Подробнее →](#)



## Аутентификация

Аутентификация (Authentication) в Laravel предоставляет простой и удобный способ проверки подлинности пользователей. С помощью встроенных функций аутентификации вы можете легко добавить систему регистрации, входа и выхода из системы на свой веб-сайт Laravel.

[Подробнее →](#)



## Авторизация

Авторизация (Authorization) в Laravel позволяет контролировать доступ пользователей к определенным ресурсам или действиям. Это позволяет легко определить, какие пользователи имеют право выполнять определенные операции в вашем приложении.

[Подробнее →](#)



## Artisan Console

Вы можете создавать миграции, запускать тесты, управлять базой данных, генерировать код и многое другое с помощью Artisan. Команды Artisan упрощают разработку, улучшают производительность и помогают взаимодействовать с вашим приложением Laravel из командной строки.

[Подробнее →](#)



## Тестирование

Встроенная система тестирования Laravel, использующая PHPUnit, обеспечивает удобные инструменты для создания и выполнения тестовых сценариев. Вы можете тестировать маршруты, контроллеры, модели и другие компоненты вашего приложения, чтобы гарантировать их работоспособность и соответствие ожиданиям.

[Подробнее →](#)

# Удобная работа с данными

Laravel имеет мощные инструменты для работы с базами данных. Он поддерживает широкий спектр СУБД, включая MySQL, MariaDB, PostgreSQL, SQL Server и SQLite. Вот несколько ключевых возможностей для работы с базой данных в Laravel:

## Eloquent ORM

Не бойтесь работать с базами данных! Laravel позволяет легко взаимодействовать с данными вашего приложения.

Создавайте модели, миграции и связи между ними в несколько простых шагов:

```
php artisan make:model Invoice --migration
```

После определения структуры модели и ее отношений, можно легко взаимодействовать с базой данных, используя мощный и выразительный синтаксис Eloquent:

```
// Создание связанной модели ...
$user->invoices()->create(['amount' => 100]);

// Обновление модели ...
$invoice->update(['amount' => 200]);

// Получение моделей ...
$invoices = Invoice::unpaid()
    ->where('amount', '>=', 100)
    ->get();

// Удобный API для взаимодействия ...
$invoices->each->pay();
```

## Миграции базы данных

Миграции в Laravel - это аналог контроля версий для вашей базы данных. Они позволяют вашей команде определить и поделиться структурой вашей базы данных:

```
// Создание таблицы "flights"
Schema::create('flights', ...);

// Установите столбец primary ключа как UUID
$table->uuid('id')->primary();

// Установите ограничение внешнего ключа
$table->foreignUuid('airline_id')
    ->constrained();
```

```
// Добавьте столбец для названия рейса  
$table->string('name');  
  
// Добавьте временные метки  
$table->timestamaps();
```

# Максимальная эффективность

Позвольте своему приложению работать с максимальной эффективностью благодаря очередям в Laravel. Независимо от того, нужно ли обрабатывать длительные задачи, отправлять уведомления или обновлять данные, очереди позволяют вам добиться максимальной пропускной способности и отзывчивости в вашем приложении.

## Job Queues

Очереди задач (Job Queues) в Laravel позволяют вам перенести медленные задачи в фоновый режим, сохраняя отзывчивость веб-запросов. Пример использования:

```
$podcast = Podcast::create(/* ... */);  
  
ProcessPodcast::dispatch($podcast)  
    ->onQueue('podcasts');
```

Вы можете запускать столько процессов очередей, сколько нужно для обработки вашей нагрузки:

```
php artisan queue:work redis --queue=podcasts
```

The screenshot shows the Laravel Horizon dashboard. On the left is a sidebar with links: Dashboard (selected), Monitoring, Metrics, Batches, Pending Jobs, Completed Jobs, Silenced Jobs, and Failed Jobs. The main area has a title "Overview". It displays the following data:

Jobs Per Minute	Jobs Past Hour	Failed Jobs Past 7 Days	Status
246	1,100	0	Active
Total Processes	Max Wait Time	Max Runtime	Max Throughput
2	-	default	default

Below this is a section titled "Current Workload" with a table:

Queue	Jobs	Processes	Wait
default	200	2	A few seconds

Finally, there is a table for supervisors:

Supervisor	Queues	Processes	Balancing
supervisor-1	default	2	Auto

## Horizon

Для удобного контроля и отслеживания очередей используйте Laravel Horizon. Horizon предоставляет красивую панель управления и конфигурацию через код для ваших очередей, работающих на Redis.

## ЧИСТЫЙ КОД

# Простые правила для вашего кода

Код должен быть понятен всем членам команды и легко читаем для разработчиков, которые могут внести изменения в него.

```
// Получаем инсайты трендов для маркетинга
$trendInsights = $this->getTrendInsights

// Запускаем кампанию с полученными данными
$campaignResults = $this->executeCampaign

// Возвращаем результаты кампании
return response()->json([
    'status' => Status::SUCCESS,
    'campaignResults' => $campaignResults
]);
```

0

## Основы

Цель этого раздела - предоставить вам практические советы по написанию чистого и понятного кода.

Здесь мы фокусируемся на конкретных советах и приемах, которые помогут сделать ваш код более чистым и понятным для других разработчиков. Мы не будем касаться глубоких тем, но сконцентрируемся на

моментах, которые могут быть применены немедленно для улучшения качества вашего кода.

Если вы хотите углубить свои знания о чистом коде, не стесняйтесь обратиться к классическим произведениям, таким как книга Роберта Мартина "Чистый код".

1

## Последовательность

Важно, чтобы стиль был единым по всему проекту.

Следуйте принципам последовательного форматирования вашего кода. Стиль кода должен соответствовать стандарту [PER](#), основанному на стандартах [PSR-1](#), [PSR-2](#) и [PSR-12](#), а также любым внутренним правилам вашей команды разработки.

Важно, чтобы стиль был единым по всему проекту.

Для автоматизации этого процесса вы можете использовать различные инструменты, такие как [Laravel Pint](#) и [PHP-CS-Fixer](#). Эти инструменты помогут поддерживать согласованный стиль кода и сделают его более читаемым и легким для понимания всеми участниками команды разработки.

```
// Плохо ✘  
class ChirpController extends Controller {
```

```
public function index (){
    $chirps = Chirp::with('user')->latest()->get();
    return view('chirps.index',[
        'chirps' => $chirps]);
}

public function update(Request $request , Chirp $chirp)
{
    $chirp->update($request->validated());

    return redirect()->route('chirps.index');
}
}
```

В этом примере кода отсутствует последовательность форматирования.

```
// Хорошо ✓
class ChirpController extends Controller
{
    public function index()
    {
        $chirps = Chirp::with('user')->latest()->get();

        return view('chirps.index', [
            'chirps' => $chirps,
        ]);
    }

    public function update(Request $request, Chirp $chirp)
    {
        $chirp->update($request->validated());

        return redirect()->route('chirps.index');
    }
}
```

2

## Именование

Имена должны быть информативными и отражать суть того, что они представляют.

Хорошие имена помогают понять код и упрощают его поддержку и развитие. Например, если вы встретите такие имена в большой области контекста, то не получите понимания о том, что происходит, а после перехода в другой участок кода вам снова придётся вникать в суть переменной или метода, что займёт много времени и сил:

```
// Переменные ✗  
$data;  
$var;  
$info;  
  
// Методы ✗  
$user->run();  
$user->handleData();  
$user->process();
```

Старайтесь использовать информативные имена, которые отражают суть того, что они представляют, например:

```
// Хорошо ✓  
$user->latestPosts();  
$user->sendEmail(...);
```

Использование сокращений может показаться удобным для быстрого написания кода, но они могут привести к путанице и усложнить поддержку кода.

Давайте рассмотрим следующий пример:

```
// Плохо ✗  
$usr = User::find($id);  
  
// Хорошо ✓  
$currentUser = User::find($userId);
```

Здесь переменная `$usr` представляет объект пользователя. Однако, сокращённое имя `$usr` не даёт понимания того, что именно хранится в этой переменной. Более ясное имя, например, `$currentUser`, немедленно указывает на её предназначение.

```
// Плохо ✗
class UsrCtrl extends Controller {
    public function f1() {
        // ...
    }
}
```

В данном примере имя класса `UsrCtrl` не информативно. Разработчику, сталкивающемуся с этим классом впервые, будет трудно понять его назначение. Название класса должно чётко отражать его функциональность, например, `ProfileController`.

```
// Хорошо ✓
class ProfileController extends Controller
{
    public function get()
    {
        // ...
    }
}
```

Теперь давайте рассмотрим пример именования с единицами измерений

```
// Плохо ✗
// Мы не знаем, что представляет собой число 100
$averageTime = 100;

// Хорошо ✓
// Мы понимаем что значение имеет величину 100мс
$averageTimeInMs = 100;
```

Другой способ справиться с этим — создать специальные объекты. Представьте, что вам нужно работать с

процентами. Что из этого верно?

```
// Плохо ✗  
$percentage = 0.5;  
$percentage = 50;
```

Встретив такую переменную, вы не сможете сказать какое значение ожидает ваше приложение. Давайте теперь воспользуемся объектом со статическим конструктором, по одному для каждой возможности.

```
class Percentage  
{  
    public static function fromInt(int $percentage): self  
    {  
        return new self($percentage);  
    }  
  
    public static function fromFloat(float $percentage): s  
    {  
        return new self($percentage * 100);  
    }  
  
    private function __construct(  
        public int $value;  
    ) {};  
}
```

Использование класса `Percentage` поясняет, что ожидается целое число.

```
// Хорошо ✓  
$percentage = Percentage::fromFloat(0.5);  
$percentage = Percentage::fromInt(50);
```

Помните, что названия должны использовать объясняющие слова, которые помогают понять их назначение. Не стесняйтесь

использовать длинные имена,  
если они ясно описывают  
сущность. Или `html`

3

## Избегайте магических чисел

Используйте именованные константы или перечисления вместо магических чисел для повышения читаемости и поддержки кода.

При написании кода важно избегать использования магических чисел, так как они могут усложнить его понимание и поддержку. Вместо этого, рекомендуется использовать именованные константы или перечисления для повышения читаемости и ясности кода.

Часто встречается ситуация, когда разработчики используют числовые значения напрямую в коде:

```
// Плохо ✗  
if ($status == 1) {  
    // ...  
}
```

В этом примере магическое число `1` используется для определения статуса активности. Однако, такой код может быть непонятным для других разработчиков, и в долгосрочной перспективе это усложняет поддержку и изменение кода.

Для решения этой проблемы лучше использовать именованные константы:

```
// Хорошо ✓  
const STATUS_ACTIVE = 1;  
  
if ($status === STATUS_ACTIVE) {  
    // ...  
}
```

Теперь код стал более понятным и поддерживаемым. При чтении такого кода сразу становится понятно, что означает статус 1.

Можно также использовать перечисления для явного определения различных значений:

```
enum Status: string  
    case ACTIVE = 'active';  
    case INACTIVE = 'inactive';  
    case ARCHIVED = 'archived';  
  
$status = Status::ACTIVE;  
  
if ($status === Status::ACTIVE) {  
    // ...  
}  
  
  
enum Status: int  
    case ACTIVE = 1;  
    case INACTIVE = 2;  
    case ARCHIVED = 3;  
  
$status = Status::ACTIVE;  
  
if ($status === Status::ACTIVE) {  
    // ...  
}
```

Такой подход делает код более читаемым и позволяет явно указать доступные значения статуса и использовать как типизированное значение в методах, например:

```
function myFunction(Status $status)
{
    // ...
}
```

Используя именованные константы или перечисления, мы делаем код более понятным и поддерживаемым, что важно для разработки масштабируемых приложений.

4

## Избегайте использования `else`

Чем меньше вложенности, тем легче понимать код.

При написании методов старайтесь избегать излишнего использования оператора `else`, так как это может сделать код менее читаемым и более сложным для поддержки. Вместо этого, предпочтительнее использовать более ясные и прямолинейные конструкции.

Давайте рассмотрим метод, который определяет, имеет ли доступ пользователь:

```
// Плохо ✘
public function isUserAllowedToAccess(User $user): bool {
    if (!$user->isBanned()) {
        if ($user->isAdmin()) {
            // Пользователь не заблокирован и является адм
            return true;
        } else {
            if ($user->isGranted(GRANT::EDIT)) {
                // Пользователь не заблокирован и имеет ра
                return true;
            } else {
                // Пользователь не заблокирован, но не явл
                return false;
            }
        }
    }
}
```

```
        }
        // Недостижимый код, так как предыдущий блок у:
        return false;
    }
} else {
    // Пользователь заблокирован
    return false;
}
}
```

Использование `else` увеличивает глубину вложенности и делает код сложнее для понимания. Чтобы сделать код более простым и понятным, лучше использовать подход с ранним возвратом результата:

```
// Хорошо ✓
public function isUserAllowedToAccess(User $user): bool
{
    if ($user->isBanned()) {
        // Пользователь заблокирован
        return false;
    }

    if ($user->isAdmin() || $user->isGranted(GRANT::EDIT))
        // Пользователь не заблокирован и является админис
        return true;
    }

    // Пользователь не заблокирован, но не является админы
    return false;
}
```

Этот подход сокращает глубину вложенности и делает код более читаемым и понятным, что облегчает его поддержку и развитие.

## Счастливый путь

Размещайте позитивные сценарии на основном уровне вложенности.

Следует стремиться к минимизации глубины вложенности кода и предпочтительно располагать позитивные сценарии выполнения функции без вложенности. Это упрощает чтение и понимание кода, делает его более структурированным и лёгким для поддержки.

```
// Плохо ✗
public function myFunction(User $user): void {
    if ($condition) {
        // много кода
    }

    throw new Exception;
}
```

В этом плохом примере позитивный сценарий выполнения функции сразу оказывается внутри условия, а исключение находится в основной части функции. Это делает код сложным для восприятия и усложняет его понимание.

```
// Хорошо ✓
public function myFunction(User $user): void
{
    if (! $condition) {
        throw new Exception;
    }

    // много кода
}
```

В хорошем примере мы сначала проверяем условие и только затем желаемое действие. Позитивный сценарий выполнения функции оказывается в основной части

функции, что делает код более читабельным и легким для понимания.

РАЗРАБОТЧИКАМ

# Советы по безопасности

Распространенные ошибки в  
коде, приводящие к уязвимостям  
безопасности в приложениях на Laravel.

0

## Основы

Минимальные меры безопасности, которые вы должны принять  
для защиты вашего приложения.

**Безопасность слишком обширная тема!**

На этой страницы рассматриваются только основные  
ошибки связанные с написанием кода, но помните,  
что безопасность – это не только код, это также и  
конфигурация сервера, сетевая безопасность,  
управление учетными данными и многое другое.

Для аудита безопасности обратите внимание  
на профессиональные компании, такие как  
[Positive Technologies](#).

Убедитесь, что ваше приложение не находится в режиме  
отладки при работе в продакшене. Для отключения  
режима отладки установите переменную окружения  
`APP_DEBUG` в значение `false`:

```
APP_DEBUG=false
```

Иначе в случае возникновения ошибки, пользователь увидит подробную информацию (включая ключи и пароли) о вашем приложении, что может быть использовано злоумышленниками для атаки

Установите безопасные разрешения на файлы и каталоги вашего приложения Laravel:

- В общем случае все каталоги Laravel должны быть настроены с максимальным уровнем разрешений [775](#),
- Неисполнимые файлы должны иметь разрешения [664](#), чтобы обеспечить безопасность и предотвратить возможные атаки.

1

## Межсайтовый скрипting (XSS)

Не дайте чужому коду проникнуть в браузер пользователя.

В шаблонизаторе [Blade](#) используются операторы вывода `{{ }}`, которые автоматически защищают вывод с помощью функции [htmlspecialchars](#) PHP.

Это необходимо для предотвращения атак XSS, при которой злоумышленник внедряет вредоносный скрипт в веб-приложение, который выполняется в браузере пользователя. Это может привести к краже сессионных cookie, перенаправлению на вредоносные сайты, изменению содержимого страницы и другим нежелательным последствиям.

Также в Blade есть возможность отображать данные без экранирования, используя синтаксис `{!! !!}`. Однако этот подход следует применять только к надежным данным, так как иначе ваше приложение станет уязвимым для атак XSS.

Пример использования небезопасного синтаксиса в шаблоне Blade:

```
{!! request()->input('somedata') !!}
```

Для обеспечения безопасности следует предпочитать следующий подход:

```
{{ request()->input('somedata') }}
```

2

## Массовое присвоение

Предотвратите несанкционированные изменения данных.

Массовое присвоение – это уязвимость, когда ORM позволяет изменять данные, которые пользователь обычно не должен иметь возможность изменять.

Рассмотрим следующий код:

```
Route::any('/profile', function (Request $request) {
    $user = $request->user();

    $user->forceFill($request->all())->save();
```

```
        return response()->json(['user' => $user]);  
    })->middleware('auth');
```

Этот маршрут позволяет пользователям изменять информацию в своем профиле.

Однако, предположим, что в таблице пользователей есть столбец `is_admin`. Вы, вероятно, не хотите, чтобы пользователь мог изменять это значение. Но код выше позволяет им изменять любые значения в их профиле, включая `is_admin`. Это уязвимость массового присвоения.

В Laravel по умолчанию встроены функции для защиты от этой уязвимости. Чтобы оставаться в безопасности:

- Ограничьте параметры, которые вы хотите обновить, используя `$request->only` или `$request->validated`, а не `$request->all`.
- Относитесь с осторожностью устанавливая значения `$guarded` и `$fillable`. В большинстве случаев, вы должны использовать `$fillable` для разрешения массового присвоения, а не `$guarded`. Однако, если вы используете `$guarded`, убедитесь, что вы добавили все поля, которые вы хотите защитить.
- Избегайте использования методов, таких как `forceFill` или `forceCreate`, которые могут обойти защиту. Однако, если вы передаете проверенный массив значений, вы можете использовать их.

3

## Загрузка файлов

Всегда проверяйте тип файла и не доверяйте пользователю определять имена файлов или пути.

Всегда проверяйте тип файла (расширение или MIME-тип), чтобы избежать выполнение удаленного кода:

```
$request->validate([
    'photo' => 'file|size:100|mimes:jpg,bmp,png'
]);
```

Атаки на выполнение удаленного кода включают загрузку вредоносных исполняемых файлов (например файлы PHP) и затем запуск их вредоносного кода, посещая URL-адрес файла (если он публичный).

По возможности старайтесь избегать обработки ZIP/XML файлов, так как они могут быть использованы для атак. Например, XXE и XEE атаки для XML и атаки на отказ в обслуживании для ZIP файлов (ZIP-бомбы).

Если ваше приложение позволяет пользовательским данным управлять путем файла для загрузки, это может привести к перезаписи критического файла или сохранению файла в неподходящем месте.

Рассмотрим следующий код:

```
Route::post('/upload', function (Request $request) {
    $request->file('file')->storeAs(
        $request->user()->id(),
        $request->input('filename')
    );

    return back();
});
```

Этот маршрут сохраняет файл в каталоге, специфическом для идентификатора пользователя. Здесь мы полагаемся на данные ввода пользователя `filename`, и это может привести к уязвимости, поскольку имя файла может быть что-то вроде `../2/filename.pdf`. Это приведет к загрузке файла в каталог пользователя с идентификатором 2, а не в каталог, соответствующий текущему вошедшему пользователю.

Чтобы исправить это, мы должны использовать функцию PHP `basename`, чтобы удалить любую информацию о директории из данных ввода `filename`:

```
Route::post('/upload', function (Request $request) {
    $request->file('file')->storeAs(
        $request->user()->id(),
        basename($request->input('filename'))
    );

    return back();
});
```

4

## Обход каталога

Не доверяйте даже имени файла, которое вы получаете от пользователя.

Обход каталога (Directory Traversal) – это атака, направленная на получение доступа к файлам путем изменения данных запроса. Это достигается за счет использования последовательностей `..` и их вариаций, а также абсолютных путей к файлам.

Если ваше приложение позволяет пользователям загружать файлы с сохранением оригинального имени файла, оно может столкнуться с уязвимостью Directory Traversal, если не проводится достаточная фильтрация входных данных для удаления информации о директории.

Давайте рассмотрим следующий пример кода:

```
Route::get('/download', function(Request $request) {
    return response()->download(
        storage_path('content/').$request->input('filename')
    );
});
```

В данном случае имя файла не очищается от информации о директории, что означает, что некорректное имя файла, например, `../../../.env`, может позволить злоумышленнику получить доступ к конфиденциальным данным вашего приложения.

Чтобы избежать этого, необходимо использовать функцию PHP `basename` для очистки информации о директории, например:

```
Route::get('/download', function(Request $request) {
    return response()->download(
        storage_path('content/').basename($request->input(
    ));
});
```

5

## SQL-инъекции

Всегда используйте привязку данных для запросов, избегайте пользовательских данных в именах столбцов и будьте

внимательны при валидации.

По умолчанию, Eloquent ORM Laravel защищает от SQL-инъекций путем параметризации запросов и использования привязок SQL. Например, рассмотрим следующий запрос:

```
use App\Models\User;  
  
User::where('email', $email)->get();
```

Приведенный выше код выполняет следующий запрос:

```
select *  
from `users`  
where `email` = ?
```

Таким образом, даже если `$email` является ненадежными данными ввода пользователя, вы защищены от атак SQL-инъекций. Однако есть несколько случаев, когда вы можете быть уязвимы к SQL-инъекциям:

## Необработанные SQL-запросы

Хотя Laravel предоставляет возможность использовать необработанные выражения запросов для создания более сложных или специфичных запросов (Например, для не поддерживаемой базы данных), следует быть осторожным и всегда использовать привязку данных. Например:

```
use Illuminate\Support\Facades\DB;  
use App\Models\User;  
  
User::whereRaw('email = "'.$request->input('email').'"')  
->get();  
  
// или так:
```

```
DB::table('users')
    ->whereRaw('email = "'.$request->input('email').'"')
    ->get();
```

Эти запросы уязвимы к инъекциям, так как не используют привязки SQL для данных полученных от пользователя. Мы можем исправить код выше, сделав следующую модификацию:

```
use App\Models\User;

User::whereRaw('email = ?', [
    $request->input('email')
])->get();
```

Мы даже можем использовать именованные привязки SQL, как показано ниже:

```
use App\Models\User;

User::whereRaw('email = :email', [
    'email' => $request->input('email')
])->get();
```

## SQL-инъекции по именам столбцов

Вы никогда не должны разрешать пользовательские данные влиять на имена столбцов, на которые ссылаются ваши запросы.

Следующие запросы могут быть уязвимы к SQL-инъекциям:

```
use App\Models\User;

User::where($request->input('colname'), 'somedata')
    ->get();

User::query()->orderBy($request->input('sortBy'))
    ->get();
```

Важно отметить, что несмотря на то, что в Laravel имеются встроенные средства защиты от SQL-инъекций, такие как об包装ование имен столбцов, некоторые базы данных могут оставаться уязвимыми из-за ограничений или конфигураций.

Всегда проверяйте пользовательский ввод для таких ситуаций, как показано ниже:

```
use App\Models\User;

$request->validate([
    'sortBy' => Rule::in(['price', 'updated_at']),
]);

User::query()
    ->orderBy($request->validated()['sortBy'])
    ->get();
```

## Правила валидации подверженные SQL-инъекциям

Некоторые механизмы валидации данных могут включать в себя возможность указания имен столбцов в базе данных. Эти правила подвержены уязвимостям SQL-инъекций, аналогично ситуации, когда SQL-инъекции направлены на имена столбцов, поскольку запросы формируются аналогичным образом.

Например, в следующем коде может существовать уязвимость:

```
use Illuminate\Validation\Rule;

$request->validate([
    'id' => Rule::unique('users')
        ->ignore($id, $request->input('colname'))
]);
```

На самом деле, этот код генерирует следующий запрос:

```
use App\Models\User;

$colname = $request->input('colname');

User::where($colname, $request->input('id'))
    ->where($colname, '<>', $id)
    ->count();
```

Поскольку имя столбца определяется пользовательским вводом, подобная ситуация эквивалентна SQL-инъекции по именам столбцов.

6

## Инъекция команд

Команды оболочки, созданные на основе ввода пользователя, могут быть опасными.

Инъекции команд – это уязвимости, связанные с возможностью выполнения команд оболочки, создаваемых на основе ввода пользователей без соответствующего экранирования.

Для иллюстрации, рассмотрим следующий код, который выполняет команду `whois` для предоставленного пользователем доменного имени:

```
public function verifyDomain(Request $request)
{
    exec('whois '.$request->input('domain'));
}
```

Этот код подвержен уязвимостям из-за недостаточного экранирования пользовательских данных. Для

исправления этой проблемы можно использовать функции PHP, такие как `escapeshellcmd` и/или `escapeshellarg`.

7

## Другие инъекции

Не используйте опасные функции на ненадежных данных.

Атаки на инъекции объектов, внедрение кода через функцию `eval` и захват переменных с использованием функции `extract` включают в себя десериализацию, выполнение кода или использование функции `extract` на ненадежных данных, полученных от пользователя.

Некоторые примеры:

```
unserialize($request->input('data'));
eval($request->input('data'));
extract($request->all());
```

В целом, избегайте передачи любых ненадежных данных в эти опасные функции.

8

## "Открытое" Перенаправление

Делайте отдельную страницу для перенаправления и предупреждайте пользователя.

Самостоятельные атаки, использующие открытое или свободное перенаправление, могут показаться не столь опасными на первый взгляд, однако они могут стать отправной точкой для атак типа "фишинг".

Рассмотрим следующий код:

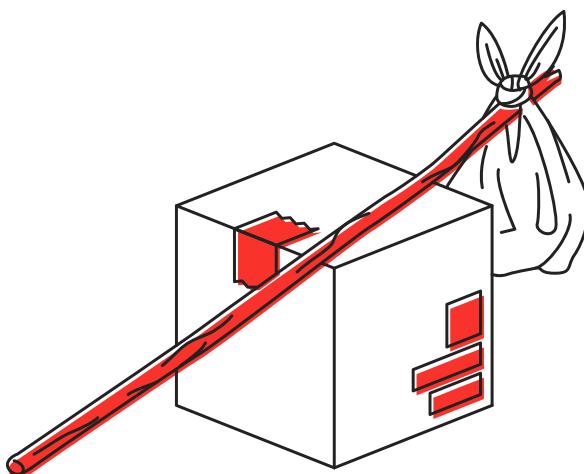
```
Route::get('/redirect', function (Request $request) {
    return redirect($request->input('url'));
});
```

Этот код осуществляет перенаправление пользователя на любой внешний URL, указанный пользовательским вводом. Это может стать уязвимостью, так как злоумышленники могут создавать видимо безопасные URL-адреса, например, <https://example.com/redirect?url=http://evil.com>. Такие URL-адреса могут быть использованы для фишинговых атак, например, подделки электронных писем о сбросе пароля, с целью заставить жертв отправить свои учетные данные на веб-сайт злоумышленника.

ДЕКЛАРАТИВНЫЙ СТИЛЬ

# Высокий уровень через коллекцию

Опирайтесь на высокие абстракции вместо применения низкоуровневых конструкций.



0

## Основы

Понимание основных принципов работы с коллекциями

Использование коллекций не является обязательным элементом, они не добавляют новую функциональность в PHP, но предоставляют удобный интерфейс для работы с перечисляемыми значениями (массивами, итераторами, генераторами).

При написании кода с использованием коллекций вы будете писать более выразительный код упрощая

множество операций с данными, таких как фильтрация, сортировка, слияние и трансформация и т.д.

Результаты запросов `Eloquent` всегда возвращаются как экземпляры `Collection`.

Когда вы пишете код в императивном стиле, вы указываете, как выполнить задачу, в то время как декларативный стиль позволяет описывать что вы хотите достичь, без явного указания шагов выполнения. Использование коллекций способствует более декларативному подходу к программированию, где вы объявляете операции над данных, а не реализуете их самостоятельно.

1

## Консистентность обработки данных

Ожидайте одинаковый порядок аргументов в вашем коде.

В стандартных функциях PHP для работы с массивами часто требуется передавать аргументы в разном порядке, что может быть запутывающим и приводить к ошибкам.

Например, функция `array_map` требует передачи callback функции первым аргументом, а массива для обработки – вторым аргументом.

В то время как функция `array_filter` требует передачи массива первым аргументом, а callback функции – вторым

аргументом. Иллюстрируем это на примере:

```
// Плохо ✗

$numbers = [1, 2, 3, 4, 5];

array_map(function ($num) {
    return $num * 2;
}, $numbers);

array_filter($numbers, function ($num) {
    return $num > 2;
});
```

Коллекции, напротив, стараются быть всегда более консистентными и предсказуемыми. Они предоставляют унифицированные методы под общим классом с похожими названиями для схожих операций, что упрощает их использование и запоминание.

Чаще всего, коллекции предоставляют методы, которые принимают `callback` функцию первым аргументом, что делает код более читаемым и понятным.

```
// Хорошо ✓

$numbers = collect([1, 2, 3, 4, 5]);

$numbers->map(function ($num) {
    return $num * 2;
});

$numbers->filter(function ($num) {
    return $num > 2;
});
```

Кроме того, методы коллекций охватывают больший спектр операций, чем стандартные функции PHP.

## Не используйте циклы

Замените циклов на методы коллекций при обработке данных.

Циклы, такие как `foreach` и `for`, широко используются для обработки данных в PHP. Однако, при использовании циклов код может стать трудночитаемым, запутанным, а следовательно подверженным ошибкам. Рассмотрим следующий распространённый пример:

```
// Плохо ✗  
$activeUsers = [];  
  
foreach ($users as $user) {  
    if ($user->isActive()) {  
        $activeUsers[] = $user;  
    }  
}
```

Здесь мы используем цикл `foreach` для фильтрации активных пользователей. Этот подход требует создания временного массива и ведет к увеличению объема кода.

При использовании коллекций доступно множество методов, таких как `filter`, `map`, `reduce`, которые позволяют заменить типичные циклы на более элегантные и понятные конструкции. Перепишем предыдущий пример, используя метод `filter`:

```
// Хорошо ✓  
$activeUsers = $users->filter(function (User $user) {  
    return $user->isActive();  
});
```

Этот код короче, проще читается и не требует создания временного массива. Метод `filter` применяет заданное условие к каждому элементу коллекции, возвращая только те, которые соответствуют критерию.

3

## Цепочка методов для обработки данных

Думай об изменениях. Как это будет работать завтра?

В программировании часто возникают задачи, требующие доработки или внесения изменения в уже существующий код. Давайте рассмотрим, как мы можем использовать методы коллекций и сравним их с использованием методов обработки массивов.

Простой пример: у нас есть набор пользователей и мы хотим отфильтровать только активных пользователей:

```
// Плохо ❌  
$activeUsers = array_filter($activeUsers, function (User $user) {  
    return $user->isActive();  
});
```

Теперь нам нужно добавить ещё один шаг: убрать администраторов из списка активных пользователей:

```
// Плохо ❌  
$activeUsers = array_filter($activeUsers, function (User $user) {  
    return $user->isActive();  
});  
  
$activeRegularUsers = array_filter($activeUsers, function
```

```
    return !$user->isAdmin();
});
```

При использовании коллекции каждый вызов метода, это отдельный шаг цепочки, который можно легко прочитать и понять:

```
// Хорошо ✓
$activeUsers = $users
    ->filter(function (User $user) {
        return $user->isActive();
    })
    ->filter(function (User $user) {
        return !$user->isAdmin();
    });
});
```

Теперь введём ещё одно условие, нам нужно отсортировать пользователей по дате регистрации:

```
// Плохо ✗
$activeUsers = array_filter($activeUsers, function (User $user) {
    return $user->isActive();
});

$activeRegularUsers = array_filter($activeUsers, function (User $user) {
    return !$user->isAdmin();
});

usort($activeRegularUsers, function (User $a, User $b) {
    return $a->created_at <=> $b->created_at;
});
```

Используя методы коллекций, мы можем добавить сортировку в цепочку методов:

```
// Хорошо ✓
$activeUsers = $users
    ->filter(function (User $user) {
        return $user->isActive();
    })
    ->filter(function (User $user) {
```

```
        return !$user->isAdmin();
    })
->sortBy('created_at');
```

Использование методов коллекций позволяет нам более явно выразить наши намерения при обработке данных, делая код более структурированным и легким для понимания.

4

## Не возвращайте примитив

Не используйте примитивы там где может потребоваться продолжение цепочки

Помимо явной цепочки вызовов, мы можем передавать промежуточные значения для дальнейшей обработки. Это может привести к избыточности кода и потере читаемости. Вместо этого, использование коллекций позволяет нам проводить цепочку операций непосредственно с объектами данных, что делает код более компактным, читаемым и эффективным.

Снова проиллюстрируем на примере:

```
// Плохо ✘

function activeUsers(): array
{
    // ...

    $activeUsers = array_filter($activeUsers, function (Us
        return $user->isActive();
    });

    $activeUsers = array_filter($activeUsers, function (Us
```

```
        return !$user->isAdmin();
    });

    usort($activeUsers, function (User $a, User $b) {
        return $a->created_at <=> $b->created_at;
    });

    return $activeUsers;
}
```

Этот метод возвращает массив, что приводит к необходимости дополнительной обработки для выполнения операций.

```
$activeUsers = activeUsers();

// Вычисление среднего возраста активных пользователей
$totalAge = 0;
foreach ($activeUsers as $user) {
    $totalAge += $user->age;
}
$averageAge = $totalAge / count($activeUsers);

// Формирование списка электронных адресов активных пользо
$emailList = [];
foreach ($activeUsers as $user) {
    $emailList[] = $user->email;
}
```

Если бы метод возвращал коллекцию, это значительно упростило бы дальнейшую обработку:

```
// Хорошо ✅
function activeUsers(): Collection
{
    // ...

    return $users
        ->filter(function (User $user) {
            return $user->isActive();
        })
        ->filter(function (User $user) {
            return !$user->isAdmin();
```

```
    })  
    ->sortBy('created_at');  
}
```

Такой подход делает дальнейшие операции более простыми и читаемыми:

// Хорошо ✓

```
$activeUsers = activeUsers();
```

```
// Вычисление среднего возраста активных пользователей  
$averageAge = $activeUsers->avg('age');
```

```
// Формирование списка электронных адресов активных пользо  
$emailList = $activeUsers->pluck('email');
```

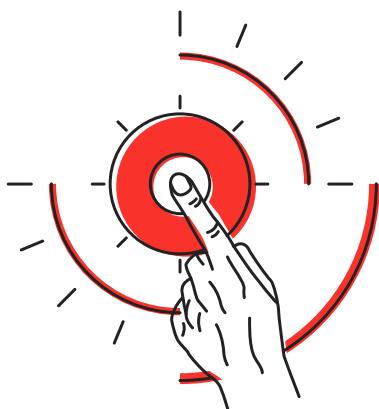


Это не только улучшает читаемость кода, но и делает его более гибким и удобным для дальнейшей обработки данных.

ЯСНОСТЬ С ПЕРВОГО ВЗГЛЯДА

## Один класс — одна задача

Каждый класс в приложении должен сосредоточиться на выполнении одной конкретной задачи или функции



0

### Основы

Что такое принцип **One Class, One Task**?

Принцип «Один класс — одна задача» (**One Class, One Task**) устанавливает требование, согласно которому каждый класс в приложении должен выполнять лишь одну конкретную задачу или функциональность. Этот подход способствует созданию более структурированного и предсказуемого кода, а также облегчает его поддержку.

В Laravel это позволяет выделить бизнес-логику в отдельные классы, что освобождает контроллеры и модели от лишних обязанностей и улучшает организацию и читаемость кода.

## Что это даст?

- **Читаемость кода:** Наличие класса, отвечающего за единую задачу, значительно упрощает понимание его назначения и поведения. Это особенно важно в рамках командной разработки или при возвращении к проекту спустя значительное время, так как открыв класс, разработчик может быстро оценить его функциональность.
- **Простота тестирования:** Изолирование логики в небольшие классы упрощает процесс тестирования. Каждый класс может быть протестирован независимо, что повышает точность и снижает время, необходимое для написания тестов.
- **Снижение сложности:** Логика приложения становится более управляемой, когда она разбивается на мелкие, специализированные части. Такой подход способствует снижению уровня сложности кода и облегчает его восприятие.
- **Легкость изменений:** Внесение изменений в код становится проще, когда изменения касаются небольших классов, каждое из которых отвечает за одну задачу. Это предотвращает возникновение неожиданностей при редактировании, так как изменения в одной части приложения не затрагивают другие компоненты.

Тем не менее, важно сохранять баланс между дроблением логики на мелкие классы и чрезмерной детализацией. Не все аспекты логики следует выносить в отдельные классы, необходимо учитывать контекст.

## Фокус на работе приложения

Использование **Actions** позволяет сосредоточиться на бизнес-логике приложения, а не на технических деталях.

Классы **Action** выполняют конкретные задачи и изолируют их от других частей приложения, что упрощает понимание кода и его поддержку. Логика, связанная с выполнением одной задачи, собирается в одном месте, что облегчает её изменение и тестирование.

Пример класса действия:

```
class GenerateReservationCode
{
    const UNAMBIGUOUS_ALPHABET = 'BCDFGHJLMNPRSTVWXYZ24567

    public function __invoke(int $characters = 7): string
    {
        do {
            $code = $this->generateCode($characters);
        } while (Reservation::where('code', $code)->exists

        return $code;
    }

    protected function generateCode(int $characters): string
    {
        return substr(str_shuffle(str_repeat(static::UNAMB
    }
}
```

Данный класс можно вызвать как функцию:

```
$generator = new GenerateReservationCode();
```

```
$reservationCode = $generator(8); // Генерация кода длиной
```

2

## Пакет Laravel Actions

Действие можно удобно запустить как объект, контроллер, фоновую задачу и консольную команду.

В экосистеме Laravel есть прекрасный пакет [Laravel Actions](#) который способствует организации кода вокруг действий. Данный пакет позволяет создавать классы действий, которые могут быть вызваны в различных контекстах, таких как контроллеры, события и консольные команды. Это обеспечивает более универсальный и гибкий код.

Пример использования:

```
class GenerateReservationCode
{
    use AsAction;

    const UNAMBIGUOUS_ALPHABET = 'BCDFGHJLMNPRSTVWXYZ24567';

    public function handle(int $characters = 7): string
    {
        do {
            $code = $this->generateCode($characters);
        } while(Reservation::where('code', $code)->exists());

        return $code;
    }

    protected function generateCode(int $characters): string
    {
        return substr(str_shuffle(str_repeat(static::UNAMB
```

```
    }  
}
```

Класс можно вызвать следующим образом:

```
GenerateReservationCode::run()
```

Если вам нужно выполнить действие в очереди, то вы также можете это сделать, например:

```
GenerateReservationCode::dispatch();
```

Вы можете узнать больше об удобстве использование действий с пакетом [Laravel Actions](#) на его [официальном сайте](#).

3

## Рекомендуемые соглашения

Помогут вам оставаться последовательными при организации вашего приложения

Для упрощения поддержки и организации кода целесообразно придерживаться ряда рекомендаций при создании классов действий.

**Начните с глагола**

Названия классов действий должны представлять собой глаголы, отражающие выполняемую задачу.

Например, если класс предназначен для отправки письма для сброса пароля, его следует назвать `SendResetPasswordEmail`.

## Используйте директорию Actions

Создайте папку `app/Actions` и сгруппируйте свои действия внутри неё по модулям. Это поможет поддерживать структуру вашего кода организованной и логичной. Например:

```
app/
└── Actions/
    ├── Authentication/
    │   ├── LoginUser.php
    │   ├── RegisterUser.php
    │   ├── ResetUserPassword.php
    │   └── SendResetPasswordEmail.php
    ├── Leads/
    │   ├── BulkRemoveLead.php
    │   ├── CreateNewLead.php
    │   ├── GetLeadDetails.php
    │   ├── MarkLeadAsCustomer.php
    │   ├── MarkLeadAsLost.php
    │   ├── RemoveLead.php
    │   ├── SearchLeadsForUser.php
    │   └── UpdateLeadDetails.php
    └── Settings/
        ├── GetUserSettings.php
        ├── UpdateUserAvatar.php
        ├── UpdateUserDetails.php
        ├── UpdateUserPassword.php
        └── DeleteUserAccount.php
└── Models/
└── ...
```

Если ваше приложение уже разделено на модули – создайте директорию `Actions` в каждом из них:

```
app/
└── Authentication/
```

```
|   └── Actions/
|   └── Models/
|   └── ...
|
└── Leads/
    ├── Actions/
    ├── Models/
    └── ...
|
└── Settings/
    ├── Actions/
    └── ...
```

Такая организация поможет вам поддерживать порядок в коде и упростит навигацию.

4

## Тестирование

Что такое принцип 'Один класс — одна задача'?

Поскольку каждый **Action** отвечает за одну задачу, его тестирование становится более простым и эффективным. Вы можете изолировать и протестировать каждое действие отдельно, что упрощает написание и выполнение тестов.

```
class GenerateReservationCodeTest extends TestCase
{
    public function testGeneratedCodeContainsOnlyAllowedCh
    {
        $code = GenerateReservationCode::run(8);

        $this->assertMatchesRegularExpression(
            '/^([BCDFGHJKLMNPRSTVWXYZ2456789]+$/',
            $code
        );
    }
}
```

```
public function testGeneratedCodeIsUrlSafe(): void
{
    $code = GenerateReservationCode::run();

    $this->assertTrue(
        filter_var($code, FILTER_VALIDATE_URL) === false
    );
}
```

А наличие четко определенных входных и выходных данных позволяют легко обнаруживать и исправлять ошибки.

- # Пролог
- # Начало работы
- # Архитектурные концепции
- # Основное
- # Погружение
- # Безопасность
- # База данных
- # Eloquent ORM
- # Тестирование
- # Пакеты

- # Пролог

- [Примечания к релизу](#)
- [Руководство по обновлению](#)
- [Рекомендации по участию](#)

- # Начало работы

- [Установка](#)
- [Конфигурация](#)
- [Структура каталогов](#)
- [Frontend](#)
- [Стартовые наборы](#)

- [Развертывание](#)

## • # Архитектурные концепции

- [Жизненный цикл запроса](#)
- [Сервис-контейнер](#)
- [Сервис-провайдеры](#)
- [Фасады](#)

## • # Основное

- [Маршрутизация](#)
- [Посредники](#)
- [CSRF Защита](#)
- [Контроллеры](#)
- [HTTP-запросы](#)
- [HTTP-ответы](#)
- [Представления](#)
- [Шаблоны Blade](#)
- [Сборка ресурсов](#)
- [Генерация URL](#)
- [Сессии](#)
- [Валидация](#)
- [Обработка ошибок](#)

- [Логирование](#)

## • # Погружение

- [Artisan коноль](#)
- [Широковещание](#)
- [Кэширование](#)
- [Коллекции](#)
- [Параллелизм](#)
- [Контекст](#)
- [Контракты](#)
- [События](#)
- [Файловое хранилище](#)
- [Помощники](#)
- [HTTP Клиент](#)
- [Локализация](#)
- [Почта](#)
- [Уведомления](#)
- [Разработка пакетов](#)
- [Процессы](#)
- [Очереди](#)
- [Ограничение скорости](#)
- [Строки](#)
- [Планировщик](#)

- # Безопасность

- Аутентификация
- Авторизация
- Верификация email
- Шифрование
- Хеширование
- Сброс пароля

- # База данных

- Начало работы
- Конструктор запросов
- Пагинация
- Миграции
- Загрузка начальных данных
- Redis

- # Eloquent ORM

- Начало работы
- Отношения
- Коллекции
- Мутаторы / Типизация

- [API Ресурсы](#)
- [Сериализация](#)
- [Фабрики](#)

- **# Тестирование**

- [Начало работы](#)
- [HTTP Тесты](#)
- [Консольные Тесты](#)
- [Браузерные Тесты](#)
- [База данных](#)
- [Имитация](#)

- **# Пакеты**

- [Breeze](#)
- [Cashier \(Stripe\)](#)
- [Cashier \(Paddle\)](#)
- [Dusk](#)
- [Envoy](#)
- [Fortify](#)
- [Folio](#)
- [Homestead](#)
- [Horizon](#)

- [Jetstream](#)
- [Mix](#)
- [Octane](#)
- [Passport](#)
- [Pennant](#)
- [Pint](#)
- [Precognition](#)
- [Prompts](#)
- [Pulse](#)
- [Reverb](#)
- [Sail](#)
- [Sanctum](#)
- [Scout](#)
- [Socialite](#)
- [Telescope](#)
- [Valet](#)
- [API Документация](#)

# Примечания к релизу

## # Схема версионирования

# Исключения

## # Политика поддержки

### # Laravel 11

# PHP 8.2

# Оптимизированная структура приложения

# Laravel Reverb

# Ограничение посекундной скорости

# Маршрутизация здоровья

# Грациозная ротация ключей шифрования

# Автоматическое изменение (rehashing) пароля

# Валидатор Prompt

# Тестирование взаимодействия с очередью

# Новые команды Artisan

# Улучшения приведения моделей

# Функция once

# Улучшенная производительность при тестировании с базами данных в памяти.

# Улучшенная поддержка MariaDB.

# Проверка баз данных и улучшенные операции со схемой

## # Схема версионирования

Laravel и другие его собственные пакеты следуют [семантическому версионированию](#). Мажорные релизы фреймворка выпускаются каждый год (примерно в первом квартале), тогда как минорные и патч-релизы могут выпускаться каждую неделю. Минорные и патч-релизы **никогда** не должны содержать критических изменений.

Ссылаясь на фреймворк Laravel или его компоненты из вашего приложения или пакета, вы всегда должны использовать ограничение версии [^11.0](#), поскольку

мажорные релизы Laravel действительно включают критические изменения. Однако мы всегда стремимся к тому, чтобы вы могли выполнить обновление до новой мажорной версии в течение дня или менее.

## Исключения

### Именованные аргументы

[Именованные аргументы](#) не подпадают под правила обратной совместимости Laravel. При необходимости мы можем переименовать аргументы функции, чтобы улучшить кодовую базу Laravel. Поэтому использовать именованные аргументы при вызове методов Laravel следует осторожно и с пониманием того, что их имена могут измениться в будущем.

## # Политика поддержки

Для всех выпусков Laravel исправления ошибок предоставляются в течение 18 месяцев, а исправления безопасности — в течение 2 лет. Для всех дополнительных библиотек, включая Lumen, только последний основной выпуск получает исправления ошибок. Кроме того, ознакомьтесь с версиями баз данных, которые [поддерживает Laravel](#).

Версия	PHP (*)	Дата релиза	Исправление ошибок до	Исправления безопасности до
9 <sup>1</sup>	8.0 -	8 февраля 2022	8 августа 2023	6 февраля 2024
	8.2			
10 <sup>2</sup>	8.1 -	14 февраля 2023	6 августа 2024	4 февраля 2025
	8.3			
11	8.2 -	12 марта 2024	3 сентября 2025	12 марта 2026
	8.3			
12	8.2 -	Q1 2025	Q3 2026	Q1 2027
	8.3			

<sup>1</sup> Окончание поддержки

<sup>2</sup> Только исправления безопасности

(\*) Поддерживаемые версии PHP

## # Laravel 11

Laravel 11 продолжает улучшения, сделанные в Laravel 10.x, представляя оптимизированную структуру приложения, ограничение скорости в секунду, маршрутизацию работоспособности, плавную ротацию ключей шифрования, улучшения тестирования очередей, почтовый транспорт [Resend](#), интеграция валидатора Prompt, новые команды Artisan и многое другое. Кроме того, был представлен Laravel Reverb, собственный масштабируемый сервер WebSocket, обеспечивающий надежные возможности работы в реальном времени для ваших приложений.

## PHP 8.2

Для Laravel 11.x требуется минимальная версия PHP 8.2.

## Оптимизированная структура приложения

*Оптимизированная структура приложения Laravel была разработана [Тейлором Отвеллом \(Taylor Otwell\)](#) и [Нуно Мадуро \(Nuno Maduro\)](#).*

Laravel 11 представляет упрощенную структуру приложений для **новых** приложений Laravel, не требующую внесения каких-либо изменений в существующие приложения. Новая структура приложения призвана обеспечить более компактный и современный интерфейс, сохраняя при этом многие концепции, с которыми разработчики Laravel уже знакомы. Ниже мы обсудим основные моменты новой структуры приложения Laravel.

## Файл начальной загрузки приложения

Файл `bootstrap/app.php` был обновлен как файл конфигурации приложения, ориентированный на код. Из этого файла вы теперь можете настроить маршрутизацию вашего приложения, посредников (middleware), поставщиков услуг, обработку исключений и многое другое. Этот файл объединяет различные высокоуровневые настройки поведения приложения, которые ранее были разбросаны по файловой структуре вашего приложения:

```
return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware) {
        //
    })
    ->withExceptions(function (Exceptions $exceptions) {
        //
    })
)->create();
```

## Поставщики услуг

Вместо структуры приложения Laravel по умолчанию, содержащей пять поставщиков услуг, Laravel 11 включает только один [AppServiceProvider](#).

Функциональность предыдущих поставщиков услуг была включена в [bootstrap/app.php](#), автоматически обрабатывается платформой или может быть помещена в [AppServiceProvider](#) вашего приложения.

Например, обнаружение событий теперь включено по умолчанию, что в значительной степени устраниет необходимость ручной регистрации событий и их прослушивателей. Однако если вам необходимо зарегистрировать события вручную, вы можете просто сделать это в [AppServiceProvider](#). Аналогично, привязки модели маршрута или шлюзы авторизации, которые вы, возможно, ранее зарегистрировали в [AuthServiceProvider](#), также могут быть зарегистрированы в [AppServiceProvider](#).

## Согласование API и широковещательная маршрутизация

Файлы маршрутов [api.php](#) и [channels.php](#) больше не присутствуют по умолчанию, поскольку многим приложениям эти файлы не требуются. Вместо этого их можно создать с помощью простых команд Artisan:

```
php artisan install:api
```

```
php artisan install:broadcasting
```

## Посредники (Middleware)

Ранее новые приложения Laravel включали девять посредников. Эти посредники выполняли различные задачи, такие как аутентификация запросов, обрезка входных строк и проверка токенов CSRF.

В Laravel 11 эти постредники были перенесены в сам фреймворк, чтобы оно не увеличивало объем структуры вашего приложения. В инфраструктуру добавлены новые методы для настройки поведения этих посредников, которые можно вызывать из файла `bootstrap/app.php` вашего приложения:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->validateCsrfTokens(
        except: ['stripe/*']
    );

    $middleware->web(append: [
        EnsureUserIsSubscribed::class,
    ])
})
```

Поскольку все промежуточное программное обеспечение можно легко настроить с помощью файла `bootstrap/app.php` вашего приложения, необходимость в отдельном классе "kernel" HTTP была устранена.

## Планирование (Scheduling)

Используя новый фасад `Schedule`, запланированные задачи теперь могут быть определены непосредственно в файле `routes/console.php` вашего приложения, что устраняет необходимость в отдельном классе "kernel" консоли:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('emails:send')->daily();
```

## Обработка исключений

Как маршрутизация и посредники, обработка исключений теперь может быть настроена из файла `bootstrap/app.php` вашего приложения вместо отдельного класса обработчика исключений, что сокращает общее количество файлов, включенных в новое приложение Laravel:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->dontReport(MissedFlightException::class);

    $exceptions->report(function (InvalidOrderException $e) {
        // ...
    });
})
```

## Базовый класс Controller

Базовый контроллер, включенный в новые приложения Laravel, был упрощен. Он больше не расширяет внутренний класс `Controller` Laravel, а трейты `AuthorizesRequests` и `ValidatesRequests` были удалены, поскольку при желании они могут быть включены в отдельные контроллеры вашего приложения:

```
<?php

namespace App\Http\Controllers;

abstract class Controller
{
    //
}
```

## Application Defaults

По умолчанию новые приложения Laravel используют SQLite для хранения базы данных, а также драйвер `database` для сеанса, кэша и очереди Laravel. Это позволяет вам приступить к разработке сразу после создания нового приложения Laravel, без необходимости устанавливать дополнительное программное обеспечение или создавать дополнительные миграции базы данных.

Кроме того, со временем драйверы `database` для этих сервисов Laravel стали достаточно надежными для производственного использования во многих контекстах приложений; поэтому они представляют собой разумный и унифицированный выбор как для локального, так и для производственного применения.

## Laravel Reverb

*Laravel Reverb* был разработан [Джо Диксоном \(Joe Dixon\)](#)

[Laravel Reverb](#) обеспечивает невероятно быструю и масштабируемую связь через WebSocket в реальном времени непосредственно в вашем приложении Laravel и обеспечивает плавную интеграцию с существующим набором инструментов трансляции событий Laravel, такими как Laravel Echo.

```
php artisan reverb:start
```

Кроме того, Reverb поддерживает горизонтальное масштабирование с помощью возможностей публикации/подписки Redis, что позволяет вам распределять трафик WebSocket между несколькими внутренними серверами Reverb, поддерживающими одно востребованное приложение.

Для получения дополнительной информации о Laravel Reverb обратитесь к полной [документации по Reverb](#).

## Ограничение посекундной скорости

Посекундное ограничение скорости было предоставлено [Тимом Макдональдом \(Tim MacDonald\)](#).

Laravel теперь поддерживает “посекундное” ограничение скорости для всех ограничителей скорости, включая ограничения для HTTP-запросов и заданий в очереди. Раньше ограничители скорости Laravel были ограничены поминутной детализацией:

```
RateLimiter::for('invoices', function (Request $request) {
    return Limit::perSecond(1);
});
```

Для получения дополнительной информации об ограничении скорости в Laravel ознакомьтесь с [документацией по ограничению скорости](#).

## Маршрутизация здоровья

Маршрутизацию работоспособности предоставил [Тейлор Отвелл \(Taylor Otwell\)](#).

Новые приложения Laravel 11 включают директиву маршрутизации `health`, которая инструктирует Laravel определить простую конечную точку проверки

работоспособности, которая может быть вызвана сторонними службами мониторинга работоспособности приложений или системами оркестрации, такими как Kubernetes. По умолчанию этот маршрут обслуживается по адресу `/up`:

```
->withRouting(  
    web: __DIR__. '/../routes/web.php',  
    commands: __DIR__. '/../routes/console.php',  
    health: '/up',  
)
```

Когда HTTP-запросы отправляются по этому маршруту, Laravel также отправляет событие `DiagnosingHealth`, позволяющее вам выполнять дополнительные проверки работоспособности, имеющие отношение к вашему приложению.

## Грациозная ротация ключей шифрования

Изящную ротацию ключей шифрования предоставил [Тейлор Отвелл \(Taylor Otwell\)](#).

Поскольку Laravel шифрует все файлы cookie, включая файлы cookie сеанса вашего приложения, по сути, каждый запрос к приложению Laravel основан на шифровании. Однако из-за этого смена ключа шифрования вашего приложения приведет к выходу всех пользователей из вашего приложения. Кроме того, расшифровка данных, зашифрованных предыдущим ключом шифрования, становится невозможной.

Laravel 11 позволяет вам определять предыдущие ключи шифрования вашего приложения в виде списка, разделенного запятыми, через переменную среды `APP_PREVIOUS_KEYS`.

При шифровании значений Laravel всегда будет использовать «текущий» ключ шифрования, который находится в переменной среды `APP_KEY`. При расшифровке значений Laravel сначала попытается использовать текущий ключ. Если расшифровка с использованием текущего ключа не удалась, Laravel попытается использовать все предыдущие ключи, пока один из ключей не сможет расшифровать значение.

Такой подход к корректному дешифрованию позволяет пользователям продолжать использовать ваше приложение непрерывно, даже если ваш ключ шифрования будет заменен.

Дополнительную информацию о шифровании в Laravel можно найти в [документации по шифрованию](#).

## Автоматическое изменение (rehashing) пароля

Автоматическое изменение (*rehashing*) пароля было предоставлено [Стивеном Рис-Кarterом \(Stephen Rees-Carter\)](#).

Алгоритм хеширования паролей в Laravel по умолчанию — bcrypt. «Рабочий коэффициент» для хешей bcrypt можно настроить с помощью файла конфигурации config/hashing.php или переменной среды BCRYPT\_ROUNDS.

Обычно рабочий коэффициент bcrypt следует увеличивать с течением времени по мере увеличения вычислительной мощности процессора/графического процессора. Если вы увеличите рабочий коэффициент bcrypt для своего приложения, Laravel теперь будет корректно и автоматически перехешировать пароли пользователей, когда пользователи проходят аутентификацию в вашем приложении.

## Валидатор Prompt

Интеграцию валидатора *Prompt* предоставил [Андреа Марко Сартори \(Andrea Marco Sartori\)](#).

[Laravel Prompts](#) — это пакет PHP для добавления красивых и удобных форм в ваши приложения командной строки с функциями браузера, включая текст-заполнитель и проверку.

Laravel Prompts поддерживает проверку ввода через замыкания:

```
$name = text(  
    label: 'What is your name?',  
    validate: fn (string $value) => match (true) {  
        strlen($value) < 3 => 'The name must be at least 3 characters.',  
        strlen($value) > 255 => 'The name must not exceed 255 characters.',  
        default => null  
    }  
)
```

Однако это может стать затруднительным при работе со многими входными данными или сложными сценариями проверки. Таким образом, в Laravel 11 вы можете использовать всю мощь [валидатора](#) Laravel при проверке ввода подсказки:

```
$name = text('What is your name?', validate: [
    'name' => 'required|min:3|max:255',
]);
```

## Тестирование взаимодействия с очередью

*Тестирование взаимодействия с очередью было предоставлено [Тейлор Отвелл \(Taylor Otwell\)](#).*

Раньше попытка проверить, было ли задание в очереди выпущено, удалено или завершилось сбоем вручную, было обременительным и требовало определения пользовательских подделок и заглушек очереди. Однако в Laravel 11 вы можете легко протестировать эти взаимодействия с очередью, используя метод `withFakeQueueInteractions`:

```
use App\Jobs\ProcessPodcast;

$job = (new ProcessPodcast)->withFakeQueueInteractions();

$job->handle();

$job->assertReleased(delay: 30);
```

Дополнительную информацию о тестировании заданий в очереди см. в [документации по очереди](#).

## Новые команды Artisan

*Команды Artisan для создания классов были предоставлены [Тейлор Отвелл \(Taylor Otwell\)](#).*

Были добавлены новые команды Artisan, позволяющие быстро создавать классы, перечисления, интерфейсы и особенности:

```
php artisan make:class  
php artisan make:enum  
php artisan make:interface  
php artisan make:trait
```

## Улучшения приведения моделей

Улучшения приведения моделей были внесены [Нуно Мадуро \(Nuno Maduro\)](#).

Laravel 11 поддерживает определение приведения модели с использованием метода, а не свойства. Это позволяет упростить и плавно определить приведение типов, особенно при использовании приведения с аргументами:

```
/**  
 * Get the attributes that should be cast.  
 *  
 * @return array<string, string>  
 */  
protected function casts(): array  
{  
    return [  
        'options' => AsCollection::using(OptionCollection::class),  
        // AsEncryptedCollection::using(OptionCollection::class),  
        // AsEnumArrayObject::using(OptionEnum::class),  
        // AsEnumCollection::using(OptionEnum::class),  
    ];  
}
```

Для получения дополнительной информации о приведении атрибутов ознакомьтесь с [документацией Eloquent](#).

## Функция once

Помощник `once` был предоставлен [Тейлор Отвелл \(Taylor Otwell\)](#) и [Нуно Мадуро \(Nuno Maduro\)](#).

Вспомогательная функция `once` выполняет заданный обратный вызов и кэширует результат в памяти на время выполнения запроса. Любые последующие вызовы функции `once` с тем же обратным вызовом будут возвращать ранее кэшированный результат:

```
function random(): int
{
    return once(function () {
        return random_int(1, 1000);
    });
}

random(); // 123
random(); // 123 (cached result)
random(); // 123 (cached result)
```

Дополнительную информацию о помощнике `once` можно найти в [документации по помощникам](#).

## Улучшенная производительность при тестировании с базами данных в памяти.

Улучшение производительности тестирования баз данных в памяти было предоставлено [Андерсом Дженбо \(Anders Jenbo\)](#)

Laravel 11 предлагает значительный прирост скорости при использовании базы данных `:memory:` SQLite во время тестирования. Для достижения этой цели Laravel теперь поддерживает ссылку на объект PDO PHP и повторно использует его при различных соединениях, часто сокращая общее время выполнения теста вдвое.

## Улучшенная поддержка MariaDB.

Улучшенную поддержку MariaDB предоставили [Йонас Штауденмейр \(Jonas Staudenmeir\)](#) и [Юлиус Кiekbusch \(Julius Kiekbusch\)](#)

Laravel 11 включает улучшенную поддержку MariaDB. В предыдущих выпусках Laravel вы могли использовать MariaDB через драйвер MySQL Laravel. Однако Laravel 11 теперь включает специальный драйвер MariaDB, который обеспечивает лучшие настройки по умолчанию для этой системы баз данных.

Дополнительную информацию о драйверах базы данных Laravel можно найти в [документации по базе данных](#).

## Проверка баз данных и улучшенные операции со схемой

*Улучшение операций со схемой и проверка базы данных предоставлено [Хафезом Дивандари \(Hafez Divandari\)](#)*

Laravel 11 предоставляет дополнительные методы работы и проверки схемы базы данных, включая встроенное изменение, переименование и удаление столбцов. Кроме того, предоставляются расширенные пространственные типы, имена схем, отличные от стандартных, и собственные методы схемы для управления таблицами, представлениями, столбцами, индексами и внешними ключами:

```
use Illuminate\Support\Facades\Schema;

$tables = Schema::getTables();
$views = Schema::getViews();
$columns = Schema::getColumns('users');
$indexes = Schema::getIndexes('users');
$foreignKeys = Schema::getForeignKeys('users');
```

# Руководство по обновлению

# Изменения, оказывающие большое влияние

# Изменения со средней степенью воздействия

# Обновление с 10.0 версии до 11.x

# Обновление зависимостей

# Структура приложения

# Аутентификация

# Кеш

# Коллекции

# База данных

# Даты

# Почта

# Пакеты

# Очереди

# Ограничение скорости

# Cashier Stripe

# Spark (Stripe)

# Passport

# Sanctum

# Telescope

# Пакет Spatie Once

# Разное

- [Обновление с 10.0 версии до 11.x](#)

## # Изменения, оказывающие большое влияние

- [Обновление зависимостей](#)

- [Структура приложения](#)

- [Типы с плавающей точкой](#)
- [Изменение столбцов](#)
- [Минимальная версия SQLite](#)
- [Обновление Sanctum](#)

## # Изменения со средней степенью воздействия

- [Carbon 3](#)
- [Рехешинг пароля](#)
- [Посекундное ограничение](#)
- [Пакет Spatie Once](#)

### Изменения с низким уровнем воздействия

- [Удаление Doctrine DBAL](#)
- [Метод casts Eloquent-модели](#)
- [Пространственные типы](#)
- [Контракт Enumerable](#)
- [Контракт UserProvider](#)
- [Контракт Authenticatable](#)

## # Обновление с 10.0 версии до 11.x

Приблизительное время обновления: 15 минут

Мы стараемся задокументировать каждое возможное изменение, которое может привести к нарушению совместимости. Поскольку некоторые из этих критических изменений находятся в малоизвестных частях фреймворка, только часть

этих изменений может повлиять на ваше приложение. Хотите сэкономить время? Вы можете использовать [Laravel Shift](#), чтобы автоматизировать процесс обновления вашего приложения.

## Обновление зависимостей

**Вероятность воздействия: высокая**

### Требуется PHP 8.2.0

Laravel теперь требует PHP версии 8.2.0 или выше.

### Требуется curl 7.34.0

HTTP-клиенту Laravel теперь требуется версия Curl 7.34.0 или выше.

## Зависимости Composer

Обновите следующие зависимости в вашем файле `composer.json`:

- `laravel/framework` to `^11.0`
- `nunomaduro/collision` to `^8.1`
- `laravel/breeze` to `^2.0` (если установлено)
- `laravel/cashier` to `^15.0` (если установлено)
- `laravel/dusk` to `^8.0` (если установлено)
- `laravel/jetstream` to `^5.0` (если установлено)
- `laravel/octane` to `^2.3` (если установлено)
- `laravel/passport` to `^12.0` (если установлено)
- `laravel/sanctum` to `^4.0` (если установлено)
- `laravel/scout` to `^10.0` (если установлено)
- `laravel/spark-stripe` to `^5.0` (если установлено)

- `laravel/telescope` to `^5.0` (если установлено)
- `livewire/livewire` to `^3.4` (если установлено)
- `inertiajs/inertia-laravel` to `^1.0` (если установлено)

Если ваше приложение использует Laravel Cashier Stripe, Passport, Sanctum, Spark Stripe или Telescope, вам необходимо опубликовать их миграции в ваше приложение. Cashier Stripe, Passport, Sanctum, Spark Stripe и Telescope **больше не загружают автоматически миграции из собственного каталога миграций.** Поэтому вам следует запустить следующую команду, чтобы опубликовать их миграции в вашем приложении:

```
php artisan vendor:publish --tag=cashier-migrations  
php artisan vendor:publish --tag=passport-migrations  
php artisan vendor:publish --tag=sanctum-migrations  
php artisan vendor:publish --tag=spark-migrations  
php artisan vendor:publish --tag=telescope-migrations
```

Кроме того, вам следует просмотреть руководства по обновлению для каждого из этих пакетов, чтобы быть в курсе любых дополнительных критических изменений:

- [Laravel Cashier Stripe](#)
- [Laravel Passport](#)
- [Laravel Sanctum](#)
- [Laravel Spark Stripe](#)
- [Laravel Telescope](#)

Если вы установили установщик Laravel вручную, вам следует обновить установщик через Composer:

```
composer global require laravel/installer:^5.6
```

Наконец, вы можете удалить зависимость Composer `doctrine/dbal`, если вы ранее добавили ее в свое приложение, поскольку Laravel больше не зависит от этого пакета.

# Структура приложения

Laravel 11 представляет новую структуру приложения с меньшим количеством файлов по умолчанию. А именно, новые приложения Laravel содержат меньше поставщиков услуг, посредников и файлов конфигурации.

Однако мы **не рекомендуем** приложениям Laravel 10, обновляющимся до Laravel 11, пытаться перенести структуру своих приложений, поскольку Laravel 11 был тщательно настроен для поддержки структуры приложений Laravel 10.

## Аутентификация

### Рехешинг пароля

#### Вероятность воздействия: Низкая

Laravel 11 автоматически перехэширует пароли вашего пользователя во время аутентификации, если «рабочий фактор» вашего алгоритма хеширования был обновлен с момента последнего хеширования пароля.

Обычно это не должно нарушать работу вашего приложения; однако, если поле «password» вашей модели `User` имеет имя, отличное от `password`, вам следует указать имя поля через свойство `authPasswordName` модели:

```
protected $authPasswordName = 'custom_password_field';
```

Так же вы можете отключить перехэширование пароля, добавив параметр `rehash_on_login` в файл конфигурации вашего приложения `config/hashing.php`:

```
'rehash_on_login' => false,
```

## Контракт UserProvider

### Вероятность воздействия: Низкая

Контракт `Illuminate\Contracts\Auth\UserProvider` получил новый метод `rehashPasswordIfRequired`. Этот метод отвечает за повторное хеширование и

сохранение пароля пользователя в хранилище при изменении коэффициента работы алгоритма хеширования приложения.

Если ваше приложение или пакет определяет класс, реализующий этот интерфейс, вам следует добавить в вашу реализацию новый метод `rehashPasswordIfRequired`. Этапонную реализацию можно найти в классе `Illuminate\Auth\EloquentUserProvider`:

```
public function rehashPasswordIfRequired(Authenticatable $user, array $credentials, I
```

## Контракт Authenticatable

### Вероятность воздействия: Низкая

Контракт `Illuminate\Contracts\Auth\Authenticatable` получил новый метод `getAuthPasswordName`. Этот метод отвечает за возврат имени столбца пароля вашего аутентифицируемого объекта.

Если ваше приложение или пакет определяет класс, реализующий этот интерфейс, вам следует добавить в вашу реализацию новый метод `getAuthPasswordName`:

```
public function getAuthPasswordName()
{
    return 'password';
}
```

Модель `User` по умолчанию, включенная в Laravel, автоматически получает этот метод, поскольку этот метод включен в трейт `Illuminate\Auth\Authenticatable`.

## Класс AuthenticationException

### Вероятность воздействия: Очень низкая

Метод `redirectTo` класса `Illuminate\Auth\AuthenticationException` теперь требует экземпляр `Illuminate\Http\Request` в качестве первого аргумента. Если вы вручную перехватываете это исключение и вызываете метод `redirectTo`, вам следует соответствующим образом обновить свой код:

```
if ($e instanceof AuthenticationException) {  
    $path = $e->redirectTo($request);  
}
```

## Кеш

### Префиксы ключей кэша

#### Вероятность воздействия: Очень низкая

Раньше, если префикс ключа кэша был определен для хранилищ кэша DynamoDB, Memcached или Redis, Laravel добавлял к префиксу `:.`. В Laravel 11 префикс ключа кэша не получает суффикс `:.`. Если вы хотите сохранить предыдущее поведение префиксом, вы можете вручную добавить суффикс `:` к префиксу ключа кэша.

## Коллекции

### Контракт Enumerable

#### Вероятность воздействия: Низкая

Метод `dump` контракта `Illuminate\Support\Enumerable` был обновлен, чтобы принимать переменный аргумент `...$args`. Если вы реализуете этот интерфейс, вам следует соответствующим образом обновить свою реализацию:

```
public function dump(...$args);
```

## База данных

### SQLite 3.26.0+

#### Вероятность воздействия: Высокая

Если ваше приложение использует базу данных SQLite, требуется SQLite 3.26.0 или более поздняя версия.

## Метод `casts` Eloquent-модели

### Вероятность воздействия: низкая

Базовый класс модели Eloquent теперь определяет метод `casts` для поддержки определения приведения атрибутов. Если одна из моделей вашего приложения определяет отношение приведения, это может конфликтовать с методом `casts`, который теперь присутствует в базовом классе модели Eloquent.

## Изменение столбцов

### Вероятность воздействия: Высокая

При изменении столбца теперь вы должны явно включать все модификаторы, которые вы хотите сохранить в определении столбца после его изменения. Любые недостающие атрибуты будут удалены. Например, чтобы сохранить атрибуты `unsigned`, `default` и `comment`, вы должны явно вызывать каждый модификатор при изменении столбца, даже если эти атрибуты были назначены столбцу при предыдущей миграции.

Например, представьте, что у вас есть миграция, в результате которой создается столбец `votes` с атрибутами `unsigned`, `default` и `comment`:

```
Schema::create('users', function (Blueprint $table) {
    $table->integer('votes')->unsigned()->default(1)->comment('The vote count');
});
```

Позже вы напишете миграцию, которая также изменит столбец на значение `nullable`:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')->nullable()->change();
});
```

В Laravel 10 эта миграция сохранит атрибуты `unsigned`, `default` и `comment` в столбце. Однако в Laravel 11 миграция теперь также должна включать все атрибуты, которые ранее были определены в столбце. В противном случае они будут удалены:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')
        ->unsigned()
        ->default(1)
        ->comment('The vote count')
        ->nullable()
        ->change();
});
```

Метод `change` не меняет индексы столбца. Поэтому вы можете использовать модификаторы индекса, чтобы явно добавлять или удалять индекс при изменении столбца:

```
// Add an index...
$table->bigIncrements('id')->primary()->change();

// Drop an index...
$table->char('postal_code', 10)->unique(false)->change();
```

Если вы не хотите обновлять все существующие миграции «изменений» в вашем приложении, чтобы сохранить существующие атрибуты столбца, вы можете просто [сократить свои миграции](#):

```
php artisan schema:dump
```

Как только ваши миграции будут завершены, Laravel «перенесет» базу данных, используя файл схемы вашего приложения, прежде чем запускать любые ожидающие миграции.

## Типы с плавающей точкой

### Вероятность воздействия: Высокая

Типы столбцов миграции `double` и `float` были переписаны, чтобы обеспечить единство во всех базах данных.

Тип столбца `double` теперь создает эквивалентный столбец `DOUBLE` без общего количества цифр и мест (цифр после десятичной точки), что является стандартным синтаксисом SQL. Поэтому вы можете удалить аргументы для `$total` и `$places`:

```
$table->double('amount');
```

Столбец типа `float` теперь создает эквивалентный столбец `FLOAT` без общего количества цифр и мест (цифр после десятичной точки), но с дополнительной спецификацией `$precision` для определения размера хранилища в виде 4-байтового столбца одинарной точности или 8-байтовый столбец двойной точности. Таким образом, вы можете удалить аргументы для `$total` и `$places` и указать необязательное значение `$precision` в соответствии с вашим желаемым значением и в соответствии с документацией вашей базы данных:

```
$table->float('amount', precision: 53);
```

Методы `unsignedDecimal`, `unsignedDouble` и `unsignedFloat` были удалены, поскольку модификатор `unsigned` для этих типов столбцов устарел в MySQL и никогда не стандартизировался в других системах баз данных. Однако, если вы хотите и дальше использовать устаревший беззнаковый атрибут для этих типов столбцов, вы можете связать метод `unsigned` с определением столбца:

```
$table->decimal('amount', total: 8, places: 2)->unsigned();
$table->double('amount')->unsigned();
$table->float('amount', precision: 53)->unsigned();
```

## Выделенный драйвер MariaDB

### Вероятность воздействия: Очень низкая

Вместо того, чтобы всегда использовать драйвер MySQL при подключении к базам данных MariaDB, Laravel 11 добавляет специальный драйвер базы данных для MariaDB.

Если ваше приложение подключается к базе данных MariaDB, вы можете обновить конфигурацию подключения новым драйвером MariaDB, чтобы в будущем воспользоваться специальными функциями MariaDB:

```
'driver' => 'mariadb',
'url' => env('DB_URL'),
'host' => env('DB_HOST', '127.0.0.1'),
```

```
'port' => env('DB_PORT', '3306'),  
// ...
```

В настоящее время новый драйвер MariaDB ведет себя как текущий драйвер MySQL, за одним исключением: метод построения схемы `uuid` создает собственные столбцы `UUID` вместо столбцов `char(36)`.

Если в ваших существующих миграциях используется метод построения схемы `uuid`, и вы решили использовать новый драйвер базы данных `mariadb`, вам следует обновить вызовы метода `uuid` в вашей миграции на `char`, чтобы избежать критических изменений или неожиданного поведения:

```
Schema::table('users', function (Blueprint $table) {  
    $table->char('uuid', 36);.  
  
    // ...  
});
```

## Пространственные типы

### Вероятность воздействия: Низкая

Типы пространственных столбцов миграции базы данных были переписаны, чтобы обеспечить единообразие во всех базах данных. Поэтому вы можете удалить методы `point`, `lineString`, `polygon`, `geometryCollection`, `multiPoint`, `multiLineString`, `multiPolygon` и `multiPolygonZ` из своих миграций и использовать вместо этого методы `geometry` или `geography`:

```
$table->geometry('shapes');  
$table->geography('coordinates');
```

Чтобы явно ограничить тип или идентификатор системы пространственной привязки для значений, хранящихся в столбце в MySQL, MariaDB и PostgreSQL, вы можете передать методу `subtype` и `srid`:

```
$table->geometry('dimension', subtype: 'polygon', srid: 0);  
$table->geography('latitude', subtype: 'point', srid: 4326);
```

Модификаторы столбцов `isGeometry` и `projection` грамматики PostgreSQL были соответственно удалены.

## Удаление Doctrine DBAL

### Вероятность воздействия: Низкая

Следующий список классов и методов, связанных с Doctrine DBAL, был удален. Laravel больше не зависит от этого пакета, и регистрация пользовательских типов Doctrines больше не требуется для правильного создания и изменения различных типов столбцов, для которых ранее требовалась пользовательские типы:

- `Illuminate\Database\Schema\Builder::$alwaysUsesNativeSchemaOperationsIfPossible` class property
- `Illuminate\Database\Schema\Builder::useNativeSchemaOperationsIfPossible()` method
- `Illuminate\Database\Connection::usingNativeSchemaOperations()` method
- `Illuminate\Database\Connection::isDoctrineAvailable()` method
- `Illuminate\Database\Connection::getDoctrineConnection()` method
- `Illuminate\Database\Connection::getDoctrineSchemaManager()` method
- `Illuminate\Database\Connection::getDoctrineColumn()` method
- `Illuminate\Database\Connection::registerDoctrineType()` method
- `Illuminate\Database\DatabaseManager::registerDoctrineType()` method
- `Illuminate\Database\PDO` directory
- `Illuminate\Database\DBAL\TimestampType` class
- `Illuminate\Database\Schema\Grammars\ChangeColumn` class
- `Illuminate\Database\Schema\Grammars\RenameColumn` class
- `Illuminate\Database\Schema\Grammars\Grammar::getDoctrineTableDiff()` method

Кроме того, больше не требуется регистрация пользовательских типов Doctrine через `dbal.types` в файле конфигурации `database` вашего приложения.

Если вы ранее использовали Doctrine DBAL для проверки вашей базы данных и связанных с ней таблиц, вы можете использовать новые собственные методы схемы Laravel (`Schema::getTables()`, `Schema::getColumns()`, `Schema::getIndexes()`, `Schema::getForeignKeys()` и т. д.).

## Устаревшие методы схемы

### Вероятность воздействия: Очень низкая

Устаревшие методы `Schema::getAllTables()`, `Schema::getAllViews()` и `Schema::getAllTypes()`, основанные на Doctrine, были удалены в пользу новых встроенных в Laravel `Schema::getTables()`, `Schema::getViews()` и `Schema::getTypes()`.

При использовании PostgreSQL и SQL Server ни один из новых методов схемы не будет принимать ссылку из трех частей (например, `database.schema.table`). Поэтому вместо этого вам следует использовать `connection()` для объявления базы данных:

```
Schema::connection('database')->hasTable('schema.table');
```

## Метод построителя схемы `getColumnType()`

### Вероятность воздействия: Очень низкая

Метод `Schema::getColumnType()` теперь всегда возвращает фактический тип данного столбца, а не эквивалентный тип Doctrine DBAL.

## Интерфейс подключения к базе данных

### Вероятность воздействия: Очень низкая

Интерфейс `Illuminate\Database\ConnectionInterface` получил новый метод `scalar`. Если вы определяете собственную реализацию этого интерфейса, вам следует добавить в вашу реализацию метод `scalar`:

```
public function scalar($query, $bindings = [], $useReadPdo = true);
```

## Даты

## Carbon 3

### Вероятность воздействия: Средняя

Laravel 11 поддерживает как Carbon 2, так и Carbon 3. Carbon — это библиотека манипулирования датами, широко используемая Laravel и пакетами во всей экосистеме. Если вы обновитесь до Carbon 3, имейте в виду, что методы `diffIn*` теперь возвращают числа с плавающей запятой и могут возвращать отрицательные значения для указания направления времени, что является существенным отличием от Carbon 2. Просмотрите [журнал изменений](<https://github.com/briannesbitt/Carbon/releases/tag/3.0.0>) и [документацию](#) Carbon для получения подробной информации о том, как обрабатывать эти и другие изменения.

## Почта

### Контракт Mailer

#### Вероятность воздействия: Очень низкая

Контракт `Illuminate\Contracts\Mail\Mailer` получил новый метод `sendNow`. Если ваше приложение или пакет вручную реализует этот контракт, вам следует добавить в свою реализацию новый метод `sendNow`:

```
public function sendNow($mailable, array $data = [], $callback = null);
```

## Пакеты

### Публикация поставщиков услуг в приложении

#### Вероятность воздействия: Очень низкая

Если вы написали пакет Laravel, который вручную публикует поставщика услуг в каталоге приложения `app/Providers` и вручную изменяет файл конфигурации приложения `config/app.php` для регистрации поставщика услуг, вам следует обновить свой пакет, чтобы использовать новый метод `ServiceProvider::addProviderToBootstrapFile`.

Метод `addProviderToBootstrapFile` автоматически добавит опубликованного вами поставщика услуг в файл `bootstrap/providers.php` приложения, поскольку массив `providers` не существует в файле конфигурации `config/app.php` в новом приложения Laravel 11.

```
use Illuminate\Support\ServiceProvider;  
  
ServiceProvider::addProviderToBootstrapFile(Provider::class);
```

## Очереди

### Интерфейс BatchRepository

**Вероятность воздействия: Очень низкая**

Интерфейс `Illuminate\Bus\BatchRepository` получил новый метод `rollBack`. Если вы реализуете этот интерфейс в своем собственном пакете или приложении, вам следует добавить в свою реализацию этот метод:

```
public function rollBack();
```

## Синхронные задания в транзакциях базы данных

**Вероятность воздействия: Очень низкая**

Раньше синхронные задания (задания, использующие драйвер очереди `sync`) выполнялись немедленно, независимо от того, было ли для параметра конфигурации `after_commit` соединения с очередью установлено значение `true` или для задания был вызван метод `afterCommit`.

В Laravel 11 синхронные задания очереди теперь будут учитывать конфигурацию соединения очереди или задания «после фиксации».

## Ограничение скорости

### Посекундное ограничение

**Вероятность воздействия: Средняя**

Laravel 11 поддерживает посекундное ограничение скорости вместо поминутной детализации. Существует множество потенциальных критических изменений, о которых вам следует знать, связанных с этим изменением.

Конструктор класса `GlobalLimit` теперь принимает секунды вместо минут. Этот класс не документирован и обычно не будет использоваться вашим приложением:

```
new GlobalLimit($attempts, 2 * 60);
```

Конструктор класса `Limit` теперь принимает секунды вместо минут. Все документированные варианты использования этого класса ограничиваются статическими конструкторами, такими как `Limit::perMinute` и `Limit::perSecond`. Однако если вы создаете экземпляр этого класса вручную, вам следует обновить приложение, чтобы предоставить секунды конструктору класса:

```
new Limit($key, $attempts, 2 * 60);
```

Свойство `decayMinutes` класса `Limit` было переименовано в `decaySeconds` и теперь содержит секунды вместо минут.

Конструкторы классов `Illuminate\Queue\Middleware\ThrottlesExceptions` и `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` теперь принимают секунды вместо минут:

```
new ThrottlesExceptions($attempts, 2 * 60);
new ThrottlesExceptionsWithRedis($attempts, 2 * 60);
```

## Cashier Stripe

### Обновление Cashier Stripe

#### Вероятность воздействия: Высокая

Laravel 11 больше не поддерживает Cashier Stripe 14.x. Поэтому вам следует обновить зависимость Laravel Cashier Stripe вашего приложения до `^15.0` в вашем файле `composer.json`.

Cashier Stripe 15.0 больше не загружает миграции автоматически из собственного каталога миграций. Вместо этого вам следует запустить следующую команду, чтобы опубликовать миграции Cashier Stripe в ваше приложение:

```
php artisan vendor:publish --tag=cashier-migrations
```

Ознакомьтесь с полным [руководством по обновлению Cashier Stripe](#), чтобы узнать о дополнительных критических изменениях.

## Spark (Stripe)

### Обновление Spark Stripe

#### Вероятность воздействия: Высокая

Laravel 11 больше не поддерживает Laravel Spark Stripe 4.x. Поэтому вам следует обновить зависимость Laravel Spark Stripe вашего приложения до [^5.0](#) в файле `composer.json`.

Spark Stripe 5.0 больше не загружает миграции автоматически из собственного каталога миграций. Вместо этого вам следует запустить следующую команду, чтобы опубликовать миграции Spark Stripe в ваше приложение:

```
php artisan vendor:publish --tag=spark-migrations
```

Ознакомьтесь с полным [руководством по обновлению Spark Stripe](#), чтобы узнать о дополнительных критических изменениях.

## Passport

### Обновление Passport

#### Вероятность воздействия: Высокая

Laravel 11 больше не поддерживает Laravel Passport 11.x. Поэтому вам следует обновить зависимость Laravel Passport вашего приложения до [^12.0](#) в вашем файле `composer.json`.

Passport 12.0 больше не загружает миграции автоматически из собственного каталога миграций. Вместо этого вам следует запустить следующую команду, чтобы опубликовать миграции Passport в вашем приложении:

```
php artisan vendor:publish --tag=passport-migrations
```

Кроме того, тип предоставления пароля по умолчанию отключен. Вы можете включить его, вызвав метод `enablePasswordGrant` в методе `boot AppServiceProvider` вашего приложения:

```
public function boot(): void
{
    Passport::enablePasswordGrant();
}
```

## Sanctum

### Обновление Sanctum

#### Вероятность воздействия: Высокая

Laravel 11 больше не поддерживает Laravel Sanctum 3.x. Поэтому вам следует обновить зависимость Laravel Sanctum вашего приложения до `^4.0` в вашем файле `composer.json`.

Sanctum 4.0 больше не загружает миграции автоматически из собственного каталога миграций. Вместо этого вам следует запустить следующую команду, чтобы опубликовать миграции Sanctum в вашем приложении:

```
php artisan vendor:publish --tag=sanctum-migrations
```

Затем в файле конфигурации вашего приложения `config/sanctum.php` вам следует обновить ссылки на посредников `authenticate_session`, `encrypt_cookies` и `validate_csrf_token` следующим образом:

```
'middleware' => [
    'authenticate_session' => Laravel\Sanctum\Http\Middleware\AuthenticateSession::class]
```

```
'encrypt_cookies' => Illuminate\Cookie\Middleware\EncryptCookies::class,  
'validate_csrf_token' => Illuminate\Foundation\Http\Middleware\ValidateCsrfToken  
],
```

## Telescope

### Обновление Telescope

#### Вероятность воздействия: Высокая

Laravel 11 больше не поддерживает Laravel Telescope 4.x. Поэтому вам следует обновить зависимость Laravel Telescope вашего приложения до [^5.0](#) в вашем файле `composer.json`.

Telescope 5.0 больше не загружает миграции автоматически из собственного каталога миграций. Вместо этого вам следует запустить следующую команду, чтобы опубликовать миграции Telescope в вашем приложении:

```
php artisan vendor:publish --tag=telescope-migrations
```

## Пакет Spatie Once

#### Вероятность воздействия: Средняя

Laravel 11 теперь предоставляет собственную функцию [once](#), чтобы гарантировать, что данное замыкание будет выполнено только один раз. Поэтому, если ваше приложение зависит от пакета `spatie/once`, вам следует удалить его из файла `composer.json` вашего приложения, чтобы избежать конфликтов.

## Разное

Мы также рекомендуем вам просмотреть изменения в [laravel/laravel репозиторий GitHub](#). Хотя многие из этих изменений не обязательны, вы можете захотеть синхронизировать эти файлы с вашим приложением. Некоторые из этих изменений будут описаны в этом руководстве по обновлению, а другие, например изменения в файлах конфигурации или комментариях, не будут рассмотрены. Вы можете легко просмотреть изменения с помощью [инструмента сравнения GitHub](#) и выбрать, какие обновления важны для вас.

# Рекомендации по участию

- # Отчеты об ошибках
- # Вопросы поддержки
- # Обсуждение разработки ядра
- # Какую ветку выбрать при запросах слияния?
- # Скомпилированные ресурсы исходников
- # Уязвимости безопасности
- # Стиль кодирования
  - # PHPDoc
  - # StyleCI
- # Нормы поведения

## # Отчеты об ошибках

Чтобы стимулировать активное сотрудничество, Laravel настоятельно рекомендует запросы на слияние, а не только отчеты об ошибках. Запросы на слияние будут рассматриваться только в том случае, если они помечены как «готовые к рассмотрению» (не в состоянии «черновик») и все тесты для новых функций пройдены. Устаревшие неактивные запросы на слияние, оставшиеся в состоянии «черновик», будут закрыты через несколько дней.

Однако, если вы отправляете отчет об ошибке, ваш тикет о проблеме должен содержать заголовок и четкое описание проблемы. Вы также должны включить как можно больше релевантной информации и образец кода, демонстрирующий проблему. Цель отчета об ошибке – упростить для вас и окружающих воспроизведение ошибки и разработать исправления.

Помните, отчеты об ошибках создаются в надежде, что другие с той же проблемой смогут сотрудничать с вами при ее решении. Не ожидайте, что отчет об ошибке автоматически сподвигнет на какие-либо действия или что другие будут прыгать, чтобы исправить ее. Создание отчета об ошибке поможет вам и другим начать работу по устранению проблемы. Если хотите внести свой вклад, то вы можете помочь, исправив [любые ошибки, перечисленные в нашем трекере тикетов](#)

[ошибок](#). Вы должны пройти аутентификацию в GitHub, чтобы просмотреть все проблемы Laravel.

Если вы заметили неправильное использование DocBlock, предупреждения PHPStan или IDE при работе с Laravel, не создавайте issue на GitHub. Вместо этого, сделайте запрос на слияние для исправления проблемы.

В управлении исходным кодом Laravel используется GitHub, и для каждого проекта есть репозитории:

- [Приложение Laravel](#)
- [Логотипы Laravel](#)
- [Laravel Breeze](#)
- [Документация Laravel](#)
- [Пакет Laravel Dusk](#)
- [Пакет Laravel Cashier Stripe](#)
- [Пакет Laravel Cashier Paddle](#)
- [Пакет Laravel Echo](#)
- [Пакет Laravel Envoy](#)
- [Пакет Laravel Folio](#)
- [Фреймворк Laravel](#)
- [Пакет Laravel Homestead \(Скрипты для сборки\)](#)
- [Пакет Laravel Horizon](#)
- [Пакет Laravel Jetstream](#)
- [Пакет Laravel Passport](#)
- [Пакет Laravel Pennant](#)
- [Пакет Laravel Pint](#)
- [Пакет Laravel Prompts](#)
- [Пакет Laravel Reverb](#)

- [Пакет Laravel Sail](#)
- [Пакет Laravel Sanctum](#)
- [Пакет Laravel Scout](#)
- [Пакет Laravel Socialite](#)
- [Пакет Laravel Telescope](#)
- [Исходники официального сайта Laravel](#)

## # Вопросы поддержки

Трекеры с тикетами проблем Laravel на GitHub не предназначены для предоставления помощи или поддержки Laravel. Вместо этого используйте один из следующих каналов:

- [Обсуждения на GitHub](#)
- [Форум Laracasts](#)
- [Форум Laravel.io](#)
- [StackOverflow](#)
- [Discord](#)
- [Larachat](#)
- [IRC](#)

## # Обсуждение разработки ядра

Вы можете предлагать новый функционал или улучшения к уже существующему поведению в репозитории фреймворка Laravel на [доске обсуждений GitHub](#). Если вы предлагаете новый функционал, то, пожалуйста, будьте готовы реализовать по крайней мере часть кода, который потребуется для его завершения.

Неформальное обсуждение ошибок, нового функционала и реализаций существующего происходит на канале `#internals` сервера [Laravel Discord](#). Тейлор Отвелл, сопровождающий Laravel, обычно присутствует на канале в будние

дни с 8:00 до 17:00 (UTC-06:00 или Америка / Чикаго) и от случая к случаю – в остальное время.

## # Какую ветку выбрать при запросах слияния?

**Все** исправления ошибок должны быть отправлены в последнюю версию, которая поддерживает исправления ошибок (на данный момент [11.x](#)). Исправления ошибок **никогда** не должны отправляться в ветку `master`, если они не исправляют функции, которые существуют только в предстоящем выпуске.

**Минорный** функционал, **полностью обратно совместимый** с текущим релизом, может быть отправлен в последнюю стабильную ветку (в настоящее время [11.x](#))..

**Мажорный** новый функционал или функционал с изменениями, приводящими к нарушению обратной совместимости, должен всегда отправляться в ветку `master`, содержащую предстоящий релиз.

## # Скомпилированные ресурсы исходников

Если вы отправляете изменение, которое повлияет на скомпилированные файлы, например, касательно файлов в `resources/css` или `resources/js` репозитория `laravel/laravel`, то не включайте в коммит эти скомпилированные файлы. Из-за большого размера они не могут быть реально рассмотрены сопровождающим. Это может быть использовано как способ внедрения вредоносного кода в Laravel. Чтобы предотвратить это, все скомпилированные файлы будут сгенерированы и включены в коммит сопровождающими Laravel.

## # Уязвимости безопасности

Если вы обнаружите уязвимость в системе безопасности Laravel, отправьте электронное письмо Тейлору Отвеллу по адресу [taylor@laravel.com](mailto:taylor@laravel.com). Все уязвимости безопасности будут незамедлительно устраниены.

## # Стиль кодирования

Laravel следует стандарту кодирования [PSR-2](#) и стандарту автозагрузки [PSR-4](#).

# PHPDoc

Ниже приведен пример валидного блока документации Laravel. Обратите внимание, что за атрибутом `@param` идут два пробела, тип аргумента, еще два пробела и, наконец, имя переменной:

```
/**
 * Register a binding with the container.
 *
 * @param string|array $abstract
 * @param \Closure|string|null $concrete
 * @param bool $shared
 * @return void
 *
 * @throws \Exception
 */
public function bind($abstract, $concrete = null, $shared = false)
```

Когда атрибуты `@param` или `@return` являются избыточными из-за использования нативных типов, их можно удалить:

```
/**
 * Execute the job.
 */
public function handle(AudioProcessor $processor): void
```

Однако, когда нативный тип является обобщенным, укажите его через использование атрибутов `@param` или `@return`:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
```

```
Attachment::fromStorage('/path/to/file'),  
];  
}
```

## StyleCI

Не волнуйтесь, если стиль вашего кода не идеален! [StyleCI](#) автоматически объединит любые исправления стиля после слияния вашего запроса с репозиторием Laravel. Это позволяет нам сосредоточиться на содержании вашего вклада, а не на стиле кода.

## # Нормы поведения

Кодекс поведения Laravel основан на кодексе поведения Ruby. О любых нарушениях кодекса поведения можно сообщить Тейлору Отвеллу ([taylor@laravel.com](mailto:taylor@laravel.com)):

- Участники будут терпимо относиться к противоположным взглядам.
- Участники должны гарантировать, что их язык и действия не содержат личных нападок и пренебрежительных личных замечаний.
- Толкуя слова и действия других, участники всегда должны исходить из добрых намерений.
- Не допускается поведение, которое можно обоснованно считать преследованием.

# Установка

## # Встречайте Laravel

# Почему именно Laravel?

## # Создание приложения Laravel

# Установка PHP и установщика Laravel

# Создание приложения

## # Начальная конфигурация

# Конфигурация на основе окружения

# Базы данных и миграции

# Конфигурация каталога

## # Локальная установка с использованием Herd

# Herd на macOS

# Herd на Windows

## # Установка Docker с использованием Sail

# Sail на macOS

# Sail на Windows

# Sail на Linux

# Выбор сервисов Sail

## # Поддержка IDE

## # Следующие шаги

# Laravel как клиент-серверный фреймворк

# Laravel в качестве сервера API

## # Встречайте Laravel

Laravel — это веб-фреймворк с выразительным и элегантным синтаксисом. Он предоставляет структуру и отправную точку для разработки приложений, позволяя сосредоточиться на создании чего-то уникального. Но пока не будем углубляться в детали.

Laravel нацелен на то, чтобы сделать процесс разработки максимально приятным, при этом предлагая мощные возможности: удобное внедрение зависимостей, выразительные абстракции для работы с базами данных, очереди и планировщик задач, поддержку модульного и интеграционного тестирования и многое другое.

Будь вы новичком в PHP, веб-фреймворках или опытным разработчиком с многолетним стажем, Laravel — это фреймворк, который будет развиваться вместе с вами. Мы поможем вам сделать первые шаги в веб-разработке или предложим способы улучшить ваши навыки. Мы с нетерпением ждём того, что вы создадите!

Новичок в Laravel? Посетите [Laravel Bootcamp](#) для практического тура по фреймворку, во время которого мы проведем вас через создание вашего первого приложения Laravel.

## Почему именно Laravel?

При разработке веб-приложений у вас есть множество инструментов и фреймворков на выбор. Однако мы уверены, что Laravel — лучший выбор для создания современных и полнофункциональных веб-приложений.

### Прогрессивный фреймворк

Мы любим называть Laravel «прогрессивным» фреймворком. Это означает, что Laravel развивается вместе с вами. Если вы начинающий разработчик, обширная библиотека документации, руководств и [видеоуроков](#) Laravel поможет вам освоить основы, не перегружая сложными концепциями.

Если вы опытный разработчик, Laravel предлагает мощные инструменты для [внедрения зависимостей](#), [модульного тестирования](#), [работы с очередями](#), [реального времени](#) и многое другое. Laravel создан для разработки профессиональных веб-приложений и способен справляться с корпоративными задачами.

### Масштабируемый фреймворк

Laravel невероятно масштабируем. Благодаря удобному для масштабирования характеру PHP и встроенной поддержке быстрых распределенных систем

кеширования, таких как Redis, горизонтальное масштабирование с Laravel очень просто. Фактически, приложения Laravel легко масштабируются для обработки сотен миллионов запросов в месяц.

Требуется экстремальное масштабирование? Такие платформы, как [Laravel Vapor](#), позволяют запускать приложение Laravel в практически неограниченном масштабе с использованием новейшей бессерверной технологии AWS.

## Фреймворк сообщества

Laravel объединяет лучшие пакеты в экосистеме PHP, чтобы предложить наиболее надёжный и удобный для разработчиков фреймворк. Более того, тысячи талантливых разработчиков со всего мира [внесли свой вклад в его развитие](#). Кто знает, возможно, вы тоже станете частью сообщества Laravel.

## # Создание приложения Laravel

### Установка PHP и установщика Laravel

Прежде чем создавать свое первое приложение Laravel, убедитесь, что на вашем локальном компьютере установлены [PHP](#), [Composer](#) и [установщик Laravel] (<https://github.com/laravel/installer>). Кроме того, вам следует установить [Node](#) и [NPM](#), чтобы вы могли скомпилировать ресурсы внешнего интерфейса вашего приложения.

Если на вашем локальном компьютере не установлены PHP и Composer, следующие команды установят PHP, Composer и установщик Laravel в macOS, Windows или Linux:

macOS      Windows PowerShell      Linux

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::
```

После выполнения одной из приведенных выше команд вам следует перезапустить сеанс терминала. Чтобы обновить PHP, Composer и установщик Laravel после их установки через [php.new](#), вы можете повторно запустить команду в своем терминале.

Если у вас уже установлены PHP и Composer, вы можете установить установщик Laravel через Composer:

```
composer global require laravel/installer
```

Чтобы получить полнофункциональный графический менеджер установки и управления PHP, посетите [Laravel Herd](#).

## Создание приложения

После установки PHP, Composer и установщика Laravel вы готовы создать новое приложение Laravel. Установщик Laravel предложит вам выбрать предпочтаемую среду тестирования, базу данных и стартовый комплект:

```
laravel new example-app
```

После создания приложения вы можете запустить локальный сервер разработки Laravel, обработчик очереди и сервер разработки Vite, используя скрипт Composer `dev`:

```
cd example-app
npm install && npm run build
composer run dev
```

После запуска сервера разработки ваше приложение будет доступно в вашем веб-браузере по адресу <http://localhost:8000>. Теперь вы готовы [продолжить свои первые шаги в мире Laravel](#). Конечно же, вы также можете [настроить базу данных](#).

Если вы хотите начать разработку вашего приложения Laravel с хорошим стартом, рассмотрите использование одного из наших [стартовых комплектов](#). Стартовые комплекти

Laravel предоставляют инфраструктуру для аутентификации как на сервере, так и на клиенте для вашего нового приложения Laravel.

## # Начальная конфигурация

Все файлы конфигурации Laravel хранятся в каталоге `config`. Каждый параметр снабжён комментариями, поэтому не стесняйтесь просматривать файлы и знакомиться с доступными вам опциями.

Laravel почти не требует дополнительной настройки сразу после установки. Вы можете начать разработку прямо сейчас! Тем не менее рекомендуется просмотреть файл `config/app.php` и его комментарии. В этом файле содержатся параметры, такие как часовой пояс и локаль, которые можно настроить в соответствии с требованиями вашего приложения.

## Конфигурация на основе окружения

Поскольку многие значения параметров конфигурации Laravel могут меняться в зависимости от того, работает ли ваше приложение на локальном компьютере или на сервере в продакшене, важные параметры конфигурации задаются в файле `.env`, который находится в корне вашего приложения.

Файл `.env` не следует добавлять в систему контроля версий вашего приложения, так как различные разработчики и серверы могут требовать разные настройки окружения. Кроме того, размещение этого файла в репозитории может представлять угрозу безопасности, если злоумышленник получит доступ к вашему репозиторио, так как конфиденциальные данные могут быть раскрыты.

Для получения дополнительной информации о конфигурации на основе файла `.env` и окружения ознакомьтесь с [документацией по конфигурации](#).

## Базы данных и миграции

После создания приложения Laravel, возможно, вам потребуется сохранить данные в базе данных. По умолчанию файл конфигурации `.env` указывает, что Laravel использует базу данных SQLite.

При создании приложения Laravel автоматически создаёт файл `database/database.sqlite` и выполняет необходимые миграции для создания таблиц базы данных.

Если вы предпочитаете использовать другой драйвер базы данных, такой как MySQL или PostgreSQL, вы можете обновить файл `.env`, чтобы указать соответствующую базу данных. Например, для использования MySQL измените переменные `DB_*` в файле `.env` следующим образом:

```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=laravel  
DB_USERNAME=root  
DB_PASSWORD=
```

Если вы выберете базу данных, отличную от SQLite, вам нужно будет создать базу данных и выполнить [миграцию базы данных](#) вашего приложения:

```
php artisan migrate
```

Если вы разрабатываете для macOS или Windows и вам необходимо установить MySQL, PostgreSQL или Redis локально, рассмотрите возможность использования [Herd Pro](#).

## Конфигурация каталога

Laravel всегда должен обслуживаться из корня веб-каталога, настроенного для вашего веб-сервера. Не пытайтесь обслуживать приложение Laravel из поддиректории, так как это может привести к открытию доступа к конфиденциальным файлам в вашем приложении.

## # Локальная установка с использованием Herd

[Laravel Herd](#) — это невероятно быстрая встроенная среда разработки Laravel и PHP для macOS и Windows. Herd включает в себя все необходимое для начала разработки на Laravel, включая PHP и Nginx.

После установки Herd вы готовы начать разработку с помощью Laravel. Herd включает инструменты командной строки для `php`, `composer`, `laravel`, `expose`, `node`, `npm` и `nvm`.

[Herd Pro](#) дополняет Herd дополнительными мощными функциями, такими как возможность создания и управления локальными базами данных MySQL, Postgres и Redis, а также локальный просмотр почты и мониторинг журналов.

## Herd на macOS

Если вы занимаетесь разработкой на macOS, вы можете загрузить установщик Herd с [веб-сайта Herd](#). Установщик автоматически загружает последнюю версию PHP и настраивает ваш Mac для постоянного запуска [Nginx](#) в фоновом режиме.

Herd для macOS использует [dnsmasq](#) для поддержки «припаркованных» каталогов. Любое приложение Laravel в припаркованном каталоге будет автоматически обслуживаться Herd. По умолчанию Herd создает припаркованный каталог в `~/Herd`, и вы можете получить доступ к любому приложению Laravel в этом каталоге в домене `.test`, используя его имя каталога.

После установки Herd самый быстрый способ создать новое приложение Laravel — использовать Laravel CLI, который входит в состав Herd:

```
cd ~/Herd
laravel new my-app
cd my-app
herd open
```

Конечно, вы всегда можете управлять своими припаркованными каталогами и другими настройками PHP через пользовательский интерфейс Herd, который можно открыть из меню Herd на панели задач.

Вы можете узнать больше о Herd, просмотрев [документацию Herd](#).

## Herd на Windows

Вы можете загрузить установщик Windows для Herd на [веб-сайте Herd](#). После завершения установки вы можете запустить Herd, чтобы завершить процесс регистрации и впервые получить доступ к пользовательскому интерфейсу Herd.

Доступ к пользовательскому интерфейсу Herd можно получить, щелкнув левой кнопкой мыши значок Herd на панели задач. Щелчок правой кнопкой мыши открывает быстрое меню с доступом ко всем инструментам, которые вам нужны ежедневно.

Во время установки Herd создает «припаркованный» каталог в вашем домашнем каталоге `%USERPROFILE%\Herd`. Любое приложение Laravel в припаркованном каталоге будет автоматически обслуживаться Herd, и вы можете получить доступ к любому приложению Laravel в этом каталоге в домене `.test`, используя его имя каталога.

После установки Herd самый быстрый способ создать новое приложение Laravel — использовать CLI Laravel, который входит в состав Herd. Для начала откройте Powershell и выполните следующие команды:

```
cd ~\Herd
laravel new my-app
cd my-app
herd open
```

Вы можете узнать больше о Herd, просмотрев [документацию Herd для Windows](#).

## # Установка Docker с использованием Sail

Мы хотим, чтобы начало работы с Laravel было максимально простым, независимо от вашей предпочтительной операционной системы. Поэтому существует несколько вариантов для разработки и запуска приложения Laravel на вашем локальном компьютере. Хотя вы можете изучить эти варианты позже, Laravel

предоставляет [Sail](#), встроенное решение для запуска приложения Laravel с использованием [Docker](#).

Docker – это инструмент для запуска приложений и служб в небольших легковесных “контейнерах”, которые не вмешиваются в установленное программное обеспечение или конфигурацию вашего локального компьютера. Это означает, что вам не нужно беспокоиться о настройке сложных инструментов разработки, таких как веб-серверы и базы данных на вашем локальном компьютере. Для начала вам нужно только установить [Docker Desktop](#).

Laravel Sail – это легкий интерфейс командной строки для взаимодействия с конфигурацией Docker по умолчанию Laravel. Sail предоставляет отличную отправную точку для создания приложения Laravel с использованием PHP, MySQL и Redis, не требуя опыта работы с Docker.

Уже являетесь экспертом по Docker? Не волнуйтесь!  
Все в Sail можно настроить с помощью файлов  
[docker-compose.yml](#), включенных в Laravel.

## Sail на macOS

Если вы разрабатываете на Mac, и [Docker Desktop](#) уже установлен, вы можете использовать простую команду в терминале, чтобы создать новое приложение Laravel. Например, чтобы создать новое приложение Laravel в каталоге с именем “example-app”, вы можете выполнить следующую команду в терминале:

```
curl -s "https://laravel.build/example-app" | bash
```

Конечно, вы можете изменить “example-app” в этом URL на что угодно, лишь бы имя приложения содержало только буквенно-цифровые символы, дефисы и подчеркивания. Директория приложения Laravel будет создана в том каталоге, из которого вы выполнили эту команду.

Установка Sail может занять несколько минут, пока контейнеры приложения Sail будут созданы на вашем локальном компьютере.

После создания приложения вы можете перейти в директорию приложения и запустить Laravel Sail. Laravel Sail предоставляет простой интерфейс командной строки для взаимодействия с конфигурацией Docker по умолчанию:

```
cd example-app  
./vendor/bin/sail up
```

После запуска контейнеров Docker приложения вам следует запустить [миграции базы данных](#):

```
./vendor/bin/sail artisan migrate
```

Наконец, вы можете получить доступ к приложению в своем веб-браузере по адресу: <http://localhost>.

Чтобы продолжить изучение Laravel Sail, ознакомьтесь с его [полной документацией](#).

## Sail на Windows

Прежде чем создавать новое приложение Laravel на вашем компьютере с Windows, убедитесь, что установлен [Docker Desktop](#). Затем убедитесь, что установлен и включен Windows Subsystem for Linux 2 (WSL2). WSL позволяет запускать исполняемые файлы Linux нативно на Windows 10. Информацию о том, как установить и включить WSL2, можно найти в документации [среды разработки Microsoft](#).

После установки и включения WSL2 убедитесь, что Docker Desktop [настроен для использования бэкенда WSL2](#).

Затем вы готовы создать свой первое приложение Laravel. Запустите [Windows Terminal](#) и начните новую сессию терминала для вашей операционной системы WSL2 Linux. Затем вы можете использовать простую команду в терминале, чтобы создать новое приложение Laravel. Например, чтобы создать новое приложение Laravel в каталоге с именем “example-app”, вы можете выполнить следующую команду в терминале:

```
curl -s https://laravel.build/example-app | bash
```

Конечно, вы можете изменить “example-app” в этом URL на что угодно, лишь бы имя приложения содержало только буквенно-цифровые символы, дефисы и подчеркивания. Директория приложения Laravel будет создана в том каталоге, из которого вы выполнили эту команду.

Установка Sail может занять несколько минут, пока контейнеры приложения Sail будут созданы на вашем локальном компьютере.

После создания приложения вы можете перейти в директорию приложения и запустить Laravel Sail. Laravel Sail предоставляет простой интерфейс командной строки для взаимодействия с конфигурацией Docker по умолчанию:

```
cd example-app
```

```
./vendor/bin/sail up
```

После запуска контейнеров Docker приложения вам следует запустить [миграции базы данных](#):

```
./vendor/bin/sail artisan migrate
```

Наконец, вы можете получить доступ к приложению в своем веб-браузере по адресу: <http://localhost>.

Чтобы продолжить изучение Laravel Sail, ознакомьтесь с его [полной документацией](#).

## Разработка в WSL2

Конечно, вам потребуется иметь возможность изменять файлы приложения Laravel, созданные в вашей установке WSL2. Для этого мы рекомендуем использовать редактор [Visual Studio Code](#) от Microsoft и их официальное расширение [Remote Development](#) для удаленной разработки.

После установки этих инструментов вы можете открыть любое приложение Laravel, выполнив команду `code .` из корневой директории вашего приложения с использованием Windows Terminal.

## Sail на Linux

Если вы разрабатываете на Linux и у вас уже установлен [Docker Compose](#), вы можете использовать простую команду терминала для создания нового приложения Laravel.

Сначала, если вы используете Docker Desktop для Linux, вам следует выполнить следующую команду. Если вы не используете Docker Desktop для Linux, этот шаг можно пропустить:

```
docker context use default
```

Затем, чтобы создать новое приложение Laravel в директории с названием "example-app", вы можете выполнить следующую команду в терминале:

```
curl -s https://laravel.build/example-app | bash
```

Конечно, вы можете изменить "example-app" в этом URL на любое другое имя, просто убедитесь, что имя приложения содержит только буквенно-цифровые символы, дефисы и подчеркивания. Директория приложения Laravel будет создана в той директории, из которой вы выполните команду.

Установка Sail может занять несколько минут, пока контейнеры приложения Sail строятся на вашем локальном компьютере.

После создания приложения вы можете перейти в директорию приложения и запустить Laravel Sail. Laravel Sail предоставляет простой интерфейс командной

строки для взаимодействия с настройками Docker по умолчанию для Laravel:

```
cd example-app  
./vendor/bin/sail up
```

После запуска контейнеров Docker приложения вам следует запустить [миграции базы данных](#):

```
./vendor/bin/sail artisan migrate
```

Наконец, вы можете получить доступ к приложению в своем веб-браузере по адресу: <http://localhost>.

Чтобы узнать больше о Laravel Sail, ознакомьтесь с его [полной документацией](#).

## Выбор сервисов Sail

При создании нового приложения Laravel через Sail вы можете использовать переменную строки запроса `with` для выбора того, какие сервисы должны быть настроены в файле `docker-compose.yml` вашего нового приложения. Доступные сервисы включают `mysql`, `pgsql`, `mariadb`, `redis`, `memcached`, `meilisearch`, `typesense`, `minio`, `selenium` и `mailpit`:

```
curl -s "https://laravel.build/example-app?with=mysql,redis" | bash
```

Если вы не укажете, какие сервисы вы хотели бы настроить, будет настроен стандартный стек из `mysql`, `redis`, `meilisearch`, `mailpit` и `selenium`.

Вы можете указать Sail установить стандартный [Devcontainer](#), добавив параметр `devcontainer` в URL:

```
curl -s "https://laravel.build/example-app?with=mysql,redis&devcontainer" | bash
```

## # Поддержка IDE

Вы можете использовать любой редактор кода при разработке приложений Laravel; однако [PhpStorm](#) предлагает обширную поддержку для Laravel и его экосистемы, включая [Laravel Pint](#).

Кроме того, поддерживаемый сообществом плагин [PhpStorm Laravel Idea](#) предлагает различные полезные дополнения для IDE, включая генерацию кода, автодополнение синтаксиса Eloquent, автодополнение правил валидации и многое другое.

## # Следующие шаги

Теперь, когда вы создали свое приложение Laravel, возможно, вам интересно, что изучить дальше. Во-первых, мы настоятельно рекомендуем ознакомиться с тем, как работает Laravel, прочитав следующую документацию:

- [Жизненный цикл запроса](#)
- [Конфигурация](#)
- [Структура директорий](#)
- [Фронтенд](#)
- [Контейнер сервисов](#)
- [Фасады](#)

Как вы планируете использовать Laravel, также определит следующие шаги на вашем пути. Существует множество способов использования Laravel, и ниже мы рассмотрим два основных варианта использования фреймворка.

Новичок в Laravel? Ознакомьтесь с [Laravel Bootcamp](#) для практического ознакомления с фреймворком,

пока мы проведем вас через создание вашего первого приложения Laravel.

## Laravel как клиент-серверный фреймворк

Laravel можно использовать как клиент-серверный фреймворк. Это означает, что вы будете использовать Laravel для маршрутизации запросов к вашему приложению и отображения интерфейса через [шаблоны Blade](#) или с помощью гибридных технологий одностраничных приложений, таких как [Inertia.js](#). Это один из наиболее распространённых способов использования Laravel.

Если вы планируете использовать Laravel в этом режиме, вам следует ознакомиться с нашей документацией по [разработке фронтенда](#), [маршрутизации](#), [представлениям](#) и [ORM Eloquent](#). Также стоит обратить внимание на пакеты сообщества, такие как [Livewire](#) и [Inertia](#), которые позволяют использовать Laravel как полноценный фреймворк и наслаждаться преимуществами одностраничных JavaScript-приложений.

Если вы используете Laravel как полноценный фреймворк, мы также рекомендуем вам изучить компиляцию CSS и JavaScript вашего приложения с помощью [Vite](#).

Если вы хотите ускорить разработку, ознакомьтесь с нашими официальными [стартовыми комплектами приложений](#).

## Laravel в качестве сервера API

Laravel также может использоваться как сервер API для одностраничных JavaScript-приложений или мобильных приложений. Например, вы можете использовать Laravel в качестве серверной части API для вашего [Next.js](#) приложения. В этом контексте Laravel может обеспечивать [авторизацию](#), а также хранение и получение данных для вашего приложения, при этом вы сможете воспользоваться мощными службами Laravel, такими как очереди, электронная почта, уведомления и многое другое.

Если вы планируете использовать Laravel таким образом, ознакомьтесь с нашей документацией по [маршрутизации](#), пакету [Laravel Sanctum](#) и [Eloquent ORM](#).

Нужен быстрый старт для создания бэкенда на Laravel и фронтенда на Next.js? Laravel Breeze предлагает [API stack](#) и [реализацию внешнего интерфейса Next.js](#), чтобы вы могли начать работу за считанные минуты.

# Конфигурирование

# Введение

# Конфигурация окружения

# Типы переменных окружения

# Получение конфигурации окружения

# Определение текущего окружения

# Шифрование файлов окружения

# Доступ к значениям конфигурации

# Кеширование конфигурации

# Публикация конфигурации

# Режим отладки

# Режим обслуживания

## # Введение

Все конфигурационные файлы фреймворка Laravel хранятся в каталоге `config`.

Каждый параметр задокументирован, поэтому не стесняйтесь просматривать эти файлы и знакомиться с доступными вам вариантами.

Конфигурационные файлы позволяют настраивать такие вещи, как информация о подключении к базе данных, информация о почтовом сервере, а также другие основные параметры, например, часовой пояс приложения и ключ шифрования.

## Команда `about`

Laravel может отображать обзор конфигурации, драйверов и среды вашего приложения с помощью команды Artisan `about`.

```
php artisan about
```

Если интересует только определенный раздел обзора приложения, вы можете фильтровать его с помощью опции `--only`:

```
php artisan about --only=environment
```

Или чтобы получить детальную информацию о значениях в определенном файле конфигурации, используйте команду `config:show` в Artisan:

```
php artisan config:show database
```

## # Конфигурация окружения

Часто бывает полезно иметь различные конфигурации в зависимости от окружения, в котором выполняется приложение. Например, по желанию можно использовать разные драйверы кеша в локальном и эксплуатационном окружении.

Чтобы упростить это, Laravel использует библиотеку [DotEnv](#) PHP. В корневом каталоге вашего нового приложения будет содержаться файл `.env.example`, определяющий множество основных переменных окружения. Этот файл будет автоматически скопирован в `.env` в процессе установки Laravel.

Файл `.env` Laravel по умолчанию содержит некоторые основные значения конфигурации, которые могут зависеть от того, работает ли ваше приложение локально или на конечном веб-сервере. Эти значения затемчитываются файлами конфигурации в каталоге `config` с помощью функции `env` Laravel.

Если вы работаете в команде, то можете не исключать файл `.env.example` и обновлять его в своем приложении. Размещая значения-заполнители в этот файл, другие разработчики в вашей команде могут четко видеть, какие переменные окружения необходимы для запуска вашего приложения.

Любая переменная в вашем файле `.env` может быть переопределена внешними переменными окружения, такими как переменные окружения уровня сервера или системы.

## Безопасность файлов окружения

Ваш файл `.env` не должен быть привязан к системе контроля версий вашего приложения, поскольку каждому разработчику / серверу, использующему ваше приложение, может потребоваться другая конфигурация окружения. Кроме того, это будет угрозой безопасности в случае, если злоумышленник получит доступ к вашему репозиторию системы управления версиями, поскольку любые конфиденциальные учетные данные будут раскрыты.

Однако, вы можете зашифровать файл среды с помощью встроенного в Laravel [шифрования среды](#). Зашифрованные файлы среды могут быть безопасно размещены в системе контроля версий.

## Дополнительные файлы окружения

Перед загрузкой переменных окружения вашего приложения Laravel определяет, была ли переменная среды `APP_ENV` предоставлена извне или указан аргумент CLI `-env`. Если это так, Laravel попытается загрузить файл `.env.[APP_ENV]`. Если он не существует, будет загружен `.env` файл по умолчанию.

## Типы переменных окружения

Все переменные в файлах `.env` обычно анализируются как строки, поэтому были созданы некоторые зарезервированные значения, позволяющие вам возвращать более широкий диапазон типов из функции `env()`:

Значение <code>.env</code>	Значение <code>env()</code>
<code>true</code>	(bool) true
<code>(true)</code>	(bool) true
<code>false</code>	(bool) false
<code>(false)</code>	(bool) false
<code>empty</code>	(string) ''
<code>(empty)</code>	(string) ''

Значение <code>.env</code>	Значение <code>env()</code>
null	(null) null
(null)	(null) null

Если вам нужно определить переменную окружения со значением, содержащим пробелы, то вы можете сделать это, заключив значение в двойные кавычки:

```
APP_NAME="My Application"
```

## Получение конфигурации окружения

Все переменные, перечисленные в этом файле, будут загружены в суперглобальную переменную `$_ENV` PHP, когда ваше приложение получит запрос. Однако вы можете использовать помощник `env()` для получения значений из переменных ваших конфигурационных файлов. Фактически, если вы просмотрите файлы конфигурации Laravel, вы заметите, что многие параметры уже используют эту функцию:

```
'debug' => env('APP_DEBUG', false),
```

Второе значение, переданное в функцию `env`, является «значением по умолчанию». Это значение будет возвращено, если для данного ключа не существует переменной окружения.

## Определение текущего окружения

Текущее окружение приложения определяется с помощью переменной `APP_ENV` из вашего файла `.env`. Вы можете получить доступ к этому значению через метод `environment` фасада App:

```
use Illuminate\Support\Facades\App;  
  
$environment = App::environment();
```

Вы также можете передать аргументы методу `environment`, чтобы определить, соответствует ли окружение переданному значению. Метод вернет `true`, если окружение соответствует любому из указанных значений:

```
if (App::environment('local')) {  
    // Локальное окружение ...  
}  
  
if (App::environment(['local', 'staging'])) {  
    // Окружение либо локальное, либо промежуточное ...  
}
```

Определение текущего окружения приложения может быть отменено путем определения переменной окружения `APP_ENV` на уровне сервера.

## Шифрование файлов окружения

Незашифрованные файлы окружения никогда не должны храниться в системе контроля версий. Однако Laravel позволяет вам зашифровать ваши файлы окружения, чтобы они безопасно могли быть добавлены в систему контроля версий вместе с остальным приложением.

### Шифрование

Для шифрования файла окружения вы можете использовать команду `env:encrypt`:

```
php artisan env:encrypt
```

Запуск команды `env:encrypt` зашифрует ваш файл `.env` и поместит зашифрованное содержимое в файл `.env.encrypted`. Ключ для расшифровки будет представлен в выводе команды и должен храниться в безопасном менеджере паролей. Если вы хотите указать свой собственный ключ шифрования, вы можете использовать опцию `--key` при вызове команды:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvobZN1aF
```

Длина указанного ключа должна соответствовать длине ключа, требуемой используемым шифром шифрования. По умолчанию Laravel использует шифр [AES-256-CBC](#), который требует ключ длиной 32 символа. Вы можете использовать любой шифр, поддерживаемый [шифратором Laravel](#), передавая опцию [--cipher](#) при вызове команды. Если у вашего приложения есть несколько файлов окружения, таких как [.env](#) и [.env.staging](#), вы можете указать имя файла окружения, который должен быть зашифрован, указав имя окружения через опцию [-env](#):

```
php artisan env:encrypt --env=staging
```

## Расшифровка

Для расшифровки файла окружения вы можете использовать команду [env:decrypt](#). Эта команда требует ключа расшифровки, который Laravel получит из переменной окружения [LARAVEL\\_ENV\\_ENCRYPTION\\_KEY](#):

```
php artisan env:decrypt
```

Ключ также может быть указан напрямую при вызове команды с помощью опции [-key](#):

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvobZN1aF
```

Когда выполняется команда [env:decrypt](#), Laravel расшифрует содержимое файла [.env.encrypted](#) и поместит расшифрованное содержимое в файл [.env](#).

Команде `env:decrypt` можно передать опцию `--cipher`, чтобы использовать пользовательский шифр шифрования:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

Если у вашего приложения есть несколько файлов окружения, таких как `.env` и `.env.staging`, вы можете указать имя файла окружения, который должен быть расшифрован, указав имя окружения через опцию `--env`:

```
php artisan env:decrypt --env=staging
```

Чтобы перезаписать существующий файл окружения, вы можете передать опцию `-force` команде `env:decrypt`:

```
php artisan env:decrypt --force
```

## # Доступ к значениям конфигурации

Вы можете легко получить доступ к своим значениям конфигурации, используя фасад `Config` или глобальную функцию `config` из любого места вашего приложения. Доступ к значениям конфигурации можно получить с помощью «точечной нотации», включающую имя файла и параметр, к которому вы хотите получить доступ. Также может быть указано значение по умолчанию, которое будет возвращено, если параметр конфигурации отсутствует:

```
use Illuminate\Support\Facades\Config;  
  
$value = Config::get('app.timezone');  
  
$value = config('app.timezone');  
  
// Получить значение по умолчанию, если значение конфигурации не существует ...  
$value = config('app.timezone', 'Asia/Seoul');
```

Чтобы установить значения конфигурации во время выполнения скрипта, вы можете вызвать метод `set` фасада `Config` или передать массив функции `config`:

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

config(['app.timezone' => 'America/Chicago']);
```

Для облегчения статического анализа фасад `Config` также предоставляет методы получения типизированной конфигурации. Если полученное значение конфигурации не соответствует ожидаемому типу, будет выдано исключение:

```
Config::string('config-key');
Config::integer('config-key');
Config::float('config-key');
Config::boolean('config-key');
Config::array('config-key');
```

## # Кеширование конфигурации

Чтобы ускорить работу вашего приложения, вы должны кешировать все конфигурационные файлы в один файл с помощью команды `config:cache` Artisan. Это объединит все конфигурационные параметры вашего приложения в один файл, который может быть быстро загружен фреймворком.

Обычно вы должны запускать команду `php artisan config:cache` как часть процесса развертывания эксплуатационного режима. Команду не следует запускать во время локальной разработки, поскольку конфигурационные параметры часто нужно будет изменять в ходе разработки вашего приложения.

После кэширования конфигурации в вашем приложении файл `.env` не будет загружен фреймворком во время запросов или команд Artisan. Поэтому функция `env` будет возвращать только внешние переменные окружения на системном уровне.

Поэтому убедитесь, что вызываете функцию `env` только из файлов конфигурации (`config`) вашего приложения. Вы можете увидеть много примеров этого, изучая файлы конфигурации по умолчанию в Laravel. Значения конфигурации могут быть получены из любого места вашего приложения с помощью функции `config`, описанной выше.

Команда `config:clear` может быть использована для очистки кэша конфигурации:

```
php artisan config:clear
```

Если вы запускаете команду `config:cache` во время процесса развертывания, убедитесь, что вы вызываете функцию `env` только из ваших файлов конфигурации. После кэширования конфигурации, файл `.env` не будет загружен, и функция `env` будет возвращать только внешние переменные окружения на системном уровне.

## # Публикация конфигурации

Большинство файлов конфигурации Laravel уже опубликованы в каталоге `config` вашего приложения; однако некоторые файлы конфигурации, такие как `cors.php` и `view.php`, не публикуются по умолчанию, поскольку большинству приложений никогда не потребуется их модифицировать.

Однако вы можете использовать Artisan-команду `config:publish` для публикации любых файлов конфигурации, которые не публикуются по умолчанию:

```
php artisan config:publish
```

```
php artisan config:publish --all
```

## # Режим отладки

Параметр `debug` в конфигурационном файле `config/app.php` определяет, сколько информации об ошибках фактически отображается конечному пользователю. По умолчанию этот параметр установлен с учетом значения переменной `APP_DEBUG` окружения, расположенной в вашем файле `.env`.

Для локальной разработки вы должны установить для переменной `APP_DEBUG` окружения значение `true`. В эксплуатационном режиме это значение всегда должно быть `false`. Если для этой переменной будет установлено значение `true`, то вы рискуете раскрыть конфиденциальные значения конфигурации конечным пользователям вашего приложения.

## # Режим обслуживания

Когда ваше приложение находится в режиме обслуживания, то для всех запросов к приложению будет отображаться специальная страница. Это позволяет легко «отключить» ваше приложение во время его обновления или технического обслуживания. Проверка режима обслуживания включена в стек посредников по умолчанию для вашего приложения. Если приложение находится в режиме обслуживания, то будет выброшено исключение `Symfony\Component\HttpFoundation\Exception\RuntimeException` с `503` кодом состояния.

Чтобы включить режим обслуживания, выполните команду `down` Artisan:

```
php artisan down
```

Если вы хотите, чтобы HTTP-заголовок `Refresh` отправлялся со всеми ответами в режиме обслуживания, вы можете указать параметр `refresh` при вызове команды `down`. Заголовок `Refresh` будет указывать браузеру автоматически обновлять страницу через указанное количество секунд:

```
php artisan down --refresh=15
```

Вы также можете передать команде `down` параметр `retry`, значение которого будет установлено в заголовке `Retry-After` HTTP<sup>1</sup>, хотя браузеры обычно игнорируют этот заголовок:

```
php artisan down --retry=60
```

## Обход режима обслуживания

Находясь в режиме обслуживания, вы можете использовать параметр `secret`, чтобы указать токен для обхода режима обслуживания:

```
php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"
```

После перевода приложения в режим обслуживания, вы можете перейти по URL-адресу приложения, с учетом этого токена, и Laravel выдаст вашему браузеру файл куки для обхода режима обслуживания:

```
https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

Если вы хотите, чтобы Laravel сгенерировал для вас секретный токен, вы можете использовать опцию `with-secret`. Секрет токен будет отображен вам, как только приложение перейдет в режим обслуживания:

```
php artisan down --with-secret
```

При доступе к этому скрытому маршруту вы будете перенаправлены на маршрут `/` приложения. Как только куки будет отправлен вашему браузеру, вы сможете просматривать приложение в обычном режиме, как если бы оно не находилось в режиме обслуживания.

Параметр `secret` режима обслуживания должен состоять из буквенно-цифровых символов и, при необходимости, тире. Вам следует избегать использования в URL-адресах символов, имеющих особое значение, таких как `? или &`.

## Режим обслуживания на нескольких серверах

По умолчанию Laravel определяет, находится ли ваше приложение в режиме обслуживания, используя файловую систему. Это означает, что для активации режима обслуживания необходимо выполнить команду `php artisan down` на каждом сервере, на котором размещено ваше приложение.

Альтернативно, Laravel предлагает метод на основе кеша для работы в режиме обслуживания. Этот метод требует запуска команды `php artisan down` только на одном сервере. Чтобы использовать этот подход, измените настройку "driver" в файле `config/app.php` вашего приложения на `cache`. Затем выберите хранилище кэша, доступное для всех ваших серверов. Это гарантирует, что статус режима обслуживания постоянно поддерживается на каждом сервере:

```
'maintenance' => [
    'driver' => 'cache',
    'store' => 'database',
],
```

## Предварительный рендеринг шаблона режима обслуживания

Если вы используете команду `php artisan down` во время развертывания, то ваши пользователи могут иногда сталкиваться с ошибками, если они обращаются к приложению во время обновления ваших зависимостей Composer или других компонентов фреймворка. Это происходит потому, для определения режима обслуживания и отображения шаблона режима обслуживания с помощью движка шаблонов должна быть загружена значительная часть фреймворка Laravel.

По этой причине Laravel позволяет в самом начале цикла запроса отобразить шаблон режима обслуживания. Этот шаблон отображается перед загрузкой любых зависимостей вашего приложения. Вы можете выполнить предварительный рендеринг шаблона по вашему выбору, используя параметр `render` команды `down`:

```
php artisan down --render="errors::503"
```

## Перенаправление запросов режима обслуживания

В режиме обслуживания Laravel будет отображать шаблон режима обслуживания для всех URL-адресов приложения, к которым пользователь попытается получить доступ. Если хотите, то вы можете указать Laravel перенаправлять все запросы на определенный URL. Это может быть выполнено с помощью параметра `redirect`. Например, вы можете перенаправить все запросы на URI `/`:

```
php artisan down --redirect=/
```

## Отключение режима обслуживания

Чтобы отключить режим обслуживания, используйте команду `up`:

```
php artisan up
```

Вы можете определить свой шаблон режима обслуживания в `resources/views/errors/503.blade.php`.

## Режим обслуживания и очереди

Пока ваше приложение находится в режиме обслуживания, поставленные в очередь задания обрабатываться не будут. Задания продолжат обрабатываться в обычном режиме после выхода приложения из режима обслуживания.

## Альтернативы режиму обслуживания

Поскольку режим обслуживания требует, чтобы ваше приложение простоявало несколько секунд, то рассмотрите альтернативы, например, [Laravel Vapor](#) и [Envoyer](#) для выполнения развертывания с нулевым временем простоя.

# Структура каталогов

- # Введение
- # Корневой каталог
- # Каталог пространства App

## # Введение

Структура приложения Laravel по умолчанию предназначена для обеспечения отличной отправной точки как для больших, так и небольших приложений. Но вы можете организовать свое приложение так, как вам нравится. Laravel почти не налагает ограничений на расположение любого конкретного класса, до тех пор, пока Composer может автоматически загружать класс.

Новичок в Laravel? Посетите [Laravel Bootcamp](#) (английский язык), чтобы практически ознакомиться с фреймворком, в то время как мы проведем вас через процесс создания вашего первого приложения на Laravel.

## # Корневой каталог

### Каталог app

Каталог `app` содержит основной код вашего приложения. Вскоре мы рассмотрим этот каталог более подробно; однако почти все классы в вашем приложении будут в этом каталоге.

## Каталог bootstrap

Каталог `bootstrap` содержит файл `app.php`, который загружает фреймворк. В этом каталоге также находится каталог `cache`, содержащий файлы, сгенерированные фреймворком для оптимизации производительности, например, файлы кеша маршрутов и служб.

## Каталог config

Каталог `config`, как следует из названия, содержит все файлы конфигурации вашего приложения. Отличная идея прочитать все эти файлы и ознакомиться со всеми доступными вам параметрами.

## Каталог database

Каталог `database` содержит миграции ваших баз данных, фабрики моделей и наполнители. При желании вы также можете использовать этот каталог для хранения SQLite БД.

## Каталог public

Каталог `public` содержит файл `index.php`, который является точкой входа для всех запросов, поступающих в ваше приложение, и конфигурирует автозагрузку. В этом каталоге также находятся ваши ресурсы, например, изображения, JavaScript и CSS.

## Каталог resources

Каталог `resources` содержит ваши шаблоны, а также ваши необработанные, нескомпилированные ресурсы, например, JavaScript или CSS.

## Каталог routes

Каталог `routes` содержит все определения маршрутов для вашего приложения. По умолчанию в Laravel входит два файла маршрутов: `web.php` и `console.php`.

Файл `web.php` содержит маршруты, которые Laravel размещает в группе посредников `web`, обеспечивающие состояние сессии, защиту CSRF и шифрование

файлов куки. Если в вашем приложении не предполагается сохранение состояния и RESTful API, то, скорее всего, все ваши маршруты будут определены в файле `web.php`.

В файле `console.php` вы можете определить все ваши анонимные консольные команды. Каждое замыкание привязано к экземпляру команды, что обеспечивает простой подход к взаимодействию с методами ввода-вывода каждой команды. Несмотря на то, что этот файл не определяет маршруты HTTP, он определяет точки входа (маршруты) в ваше консольное приложение. Вы также можете запланировать задачи в файле `console.php`.

При желании вы можете установить дополнительные файлы маршрутов для маршрутов API (`api.php`) и каналов вещания (`channels.php`) с помощью команд Artisan `install:api` и `install:broadcasting`.

Файл `api.php` содержит маршруты, которые не сохраняют состояние, поэтому запросы, поступающие в приложение через эти маршруты, предназначены для аутентификации через токены и не будут иметь доступа в состояние сеанса.

В файле `channels.php` вы можете зарегистрировать все каналы трансляции событий, которые поддерживает ваше приложение.

## Каталог `storage`

Каталог `storage` содержит ваши журналы (логи), скомпилированные шаблоны Blade, файлы сессий, кеша и другие файлы, созданные фреймворком. Этот каталог разделен на каталоги `app`, `framework`, и `logs`. Каталог `app` может использоваться для хранения любых файлов, созданных вашим приложением. Каталог `framework` используется для хранения файлов и кешей, сгенерированных фреймворком. Наконец, каталог `logs` содержит файлы журнала вашего приложения.

Каталог `storage/app/public` может использоваться для хранения файлов, созданных пользователями, таких как аватары профиля, которые должны быть общедоступными. Вы должны создать символическую ссылку (ярлык) в `public/storage`, которая указывает на этот каталог. Вы можете создать ссылку, используя команду `php artisan storage:link` Artisan.

## Каталог tests

Каталог `tests` содержит ваши автоматизированные тесты. Примеры модульного [Pest](#) или [PHPUnit](#) и функционального тестов предоставляются из коробки. Каждый тестовый класс должен иметь суффикс `Test`. Вы можете запускать свои тесты с помощью команд `/vendor/bin/pest` или `/vendor/bin/phpunit`. Или, если вы хотите более подробное и красивое отображение результатов ваших тестов, вы можете запускать свои тесты с помощью команды `php artisan test` Artisan.

## Каталог vendor

Каталог `vendor` содержит ваши [Composer](#)-зависимости.

## # Каталог пространства App

Большая часть вашего приложения находится в каталоге `app`. По умолчанию этот каталог находится в пространстве имен `App` и автоматически загружается Composer с использованием [автозагрузчика стандарта PSR-4](#).

По умолчанию каталог `app` содержит каталоги `Http`, `Models` и `Providers`. Однако со временем внутри каталога приложения будет создано множество других каталогов, поскольку вы используете команды `make` Artisan для создания классов. Например, каталог `app\Console` не будет существовать до тех пор, пока вы не выполните команду Artisan `make:command` для создания класса команды.

Каталоги `Console` и `Http` более подробно описаны в соответствующих разделах ниже, но думайте о каталогах `Console` и `Http` как о предоставлении API для ядра вашего приложения. Протокол HTTP и CLI являются механизмами взаимодействия с вашим приложением, но фактически не содержат логики приложения. Другими словами, это два способа подачи команд вашему приложению. Каталог `Console` содержит все ваши команды Artisan, а каталог `Http` содержит ваши контроллеры, посредники (middleware) и запросы.

Многие классы в каталоге `app` могут быть созданы с помощью команд Artisan. Чтобы просмотреть доступные команды, выполните команду `php artisan list make` в консоли.

## Каталог пространства Broadcasting

Каталог `Broadcasting` содержит все классы широковещательных каналов для вашего приложения. Эти классы генерируются с помощью команды `make:channel`. Этот каталог не существует по умолчанию, но будет создан для вас, когда вы создадите свой первый канал. Чтобы узнать больше о каналах, ознакомьтесь с документацией по [трансляции событий](#).

## Каталог пространства Console

Каталог `Console` содержит все пользовательские команды Artisan для вашего приложения. Эти команды могут быть сгенерированы с помощью команды `make:command`.

## Каталог пространства Events

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команды `event:generate` или `make:event` Artisan. В каталоге `Events` находятся [классы событий](#). События могут использоваться для предупреждения других частей вашего приложения о том, что произошло определенное действие, обеспечивая большую гибкость.

## Каталог пространства Exceptions

Каталог `Exceptions` содержит все пользовательские исключения для вашего приложения. Эти исключения могут быть созданы с помощью команды `make:exception`.

## Каталог пространства Http

Каталог `Http` содержит ваши контроллеры, посредники и запросы форм.

Практически вся логика обработки запросов, поступающих в ваше приложение, будет размещена в этом каталоге.

## Каталог пространства Jobs

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команду `make:job` Artisan. В каталоге `Jobs` находятся [планировщики заданий](#) вашего приложения. Задания могут быть поставлены в очередь вашим приложением или выполняться синхронно в рамках жизненного цикла текущего запроса. Задания, которые выполняются синхронно во время текущего запроса, иногда называют «командами», поскольку они являются реализацией [шаблона команды](#).

## Каталог пространства Listeners

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команды `event:generate` или `make:listener` Artisan. Каталог `Listeners` содержит классы, которые обрабатывают ваши [события](#). Слушатели событий получают экземпляр события и выполняют логику в ответ на запускаемое событие. Например, событие `UserRegistered` может обрабатываться слушателем `SendWelcomeEmail`.

## Каталог пространства Mail

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команду `make:mail` Artisan. Каталог `Mail` содержит все ваши [классы для работы с электронными письмами](#), отправляемыми вашим приложением. Почтовые объекты позволяют вам инкапсулировать всю логику создания электронной почты в один простой класс, который может быть отправлен с помощью метода `Mail::send`.

## Каталог пространства Models

Каталог `Models` содержит все ваши [классы моделей Eloquent](#). Laravel содержит

библиотеку Eloquent ORM, предоставляющую красивую и простую реализацию ActiveRecord для работы с вашей базой данных. Каждая таблица базы данных имеет соответствующую «Модель», которая используется для взаимодействия с этой таблицей. Модели позволяют запрашивать данные в таблицах, а также вставлять новые записи в таблицу.

## Каталог пространства Notifications

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команду `make:notification` Artisan. Каталог `Notifications` содержит все «транзакционные» [уведомления](#), которые отправляются вашим приложением. Например, простые уведомления о событиях, которые происходят в вашем приложении. Уведомления Laravel позволяют абстрагироваться от отправки уведомлений по различным драйверам, таким как электронная почта, Slack, SMS, или сохранение в базе данных.

## Каталог пространства Policies

Этот каталог не существует по умолчанию, но будет создан для вас, если вы выполните команду `make:policy` Artisan. Каталог `Policies` содержит [классы политик авторизации](#) вашего приложения. Политики используются для определения того, может ли пользователь выполнить определенное действие с ресурсом.

## Каталог пространства Providers

Каталог `Providers` содержит всех [поставщиков служб](#) вашего приложения. Поставщики служб загружают ваше приложение, связывая службы в контейнере служб, регистрируя события или выполняя любые другие задачи для подготовки вашего приложения к входящим запросам.

В новом приложении Laravel этот каталог уже будет содержать `AppServiceProvider`. При необходимости вы можете добавить свои собственные провайдеры в этот каталог.

## Каталог пространства Rules

Этот каталог не существует по умолчанию, но будет создан для вас, если вы

выполните команду `make:rule` Artisan. Каталог `Rules` содержит пользовательские объекты правил валидации вашего приложения. Правила используются для инкапсуляции сложной логики проверки в простой объект. Для получения дополнительной информации ознакомьтесь с [документацией по валидации](#).

# Frontend

## # Введение

## # Использование PHP

# PHP и Blade

# Livewire

# Стартовые наборы

## # Использование Vue / React

# Inertia

# Стартовые наборы

## # Сборка ресурсов (Bundling Assets)

## # Введение

Laravel – это бэкенд-фреймворк, предоставляющий все необходимые функции, такие как [маршрутизация](#), [валидация](#), [кэширование](#), [очереди](#), [хранение файлов](#) и другие, для создания современных веб-приложений. Однако мы считаем, что важно предложить разработчикам прекрасный опыт работы с полным стеком, включая мощные подходы для создания фронтенда вашего приложения.

Существует два основных способа разработки фронтенда при создании приложения на Laravel, и выбор подхода зависит от того, хотите ли вы создать фронтенд на основе PHP или с помощью JavaScript-фреймворков, таких как Vue и React. Ниже мы рассмотрим оба этих варианта, чтобы вы могли принять взвешенное решение относительно наилучшего подхода к разработке фронтенда для вашего приложения.

## # Использование PHP

### PHP и Blade

В прошлом большинство PHP-приложений отображали HTML в браузере, используя простые HTML-шаблоны с вкрапленными PHP операторами `echo`, которые выводили

данные, полученные из базы данных во время запроса:

```
<div>
    <?php foreach ($users as $user): ?>
        Привет, <?php echo $user->name; ?> <br />
    <?php endforeach; ?>
</div>
```

В Laravel такой подход к отображению HTML все еще можно реализовать с использованием [шаблонов \(views\)](#) и [Blade](#). Blade – это крайне легкий язык шаблонов, который предоставляет удобный и краткий синтаксис для отображения данных, итерации по данным и многоего другого:

```
<div>
    @foreach ($users as $user)
        Привет, {{ $user->name }} <br />
    @endforeach
</div>
```

При построении приложений в таком стиле, отправка форм и другие взаимодействия со страницей обычно влекут получение полностью нового HTML-документа от сервера, и страница полностью перерисовывается браузером. Даже сегодня многие приложения могут идеально подходить для построения своего фронтенда с использованием простых шаблонов Blade.

## Растущие ожидания

Однако с увеличением требований пользователей к веб-приложениям многие разработчики сталкиваются с необходимостью создания более динамичных интерфейсов с более отточенным взаимодействием. В связи с этим некоторые разработчики предпочитают создавать пользовательский интерфейс своего приложения с использованием JavaScript-фреймворков, таких как Vue и React.

Другие, предпочитая оставаться верными языку бэкенда, с которым они знакомы, разработали решения, которые позволяют создавать современные пользовательские интерфейсы веб-приложений, в основном используя свой предпочтаемый язык бэкенда. Например, в экосистеме [Rails](#) это подстегнуло создание таких библиотек, как [Turbo Hotwire](#), и [Stimulus](#).

В экосистеме Laravel необходимость создания современных и динамичных интерфейсов с использованием преимущественно PHP привела к появлению [Laravel Livewire](#) и [Alpine.js](#).

## Livewire

[Laravel Livewire](#) это платформа для создания фронтенда на базе Laravel, который выглядит динамичными, современными и живыми, как фронтенд, созданные с использованием современных фреймворков JavaScript, таких как Vue и React.

При использовании Livewire вы создаете “компоненты” Livewire, которые отображают отдельные части вашего пользовательского интерфейса и предоставляют методы и данные, которые можно вызывать и с которыми можно взаимодействовать из фронтенда вашего приложения. Например, простой компонент “Counter” может выглядеть следующим образом:

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
    {
        return view('livewire.counter');
    }
}
```

И соответствующий шаблон для счетчика будет выглядеть следующим образом::

```
<div>
    <button wire:click="increment">+</button>
```

```
<h1>{{ $count }}</h1>
</div>
```

Как видите, Livewire позволяет вам добавлять новые атрибуты HTML, такие как `wire:click`, которые соединяют фронтенд и бэкенд вашего приложения Laravel. Кроме того, вы можете отображать текущее состояние компонента с использованием простых выражений Blade.

Для многих Livewire стал революцией в разработке фронтенда с использованием Laravel, позволяя оставаться в зоне комфорта Laravel при создании современных динамичных веб-приложений. Обычно разработчики, использующие Livewire, также используют [Alpine.js](#) для добавления JavaScript только там, где это необходимо, например, для отображения окна диалога.

Если вы новичок в Laravel, мы рекомендуем ознакомиться с основами использования [шаблонов](#) и [Blade](#). Затем обратитесь к официальной [документации Laravel Livewire](#), чтобы узнать, как поднять свое приложение на новый уровень с помощью интерактивных компонентов Livewire.

## Стартовые наборы

Если вы хотите создать свой интерфейс с использованием PHP и Livewire, вы можете использовать наши [стартовые наборы](#) Breeze или Jetstream, чтобы ускорить разработку вашего приложения. Оба этих стартовых комплекта представляют собой основу бэкэнд- и фронтэнд-потока аутентификации вашего приложения с помощью [Blade](#) и [Tailwind](#), так что вы можете просто приступить к реализации своей следующей большой идеи.

## # Использование Vue / React

Хотя можно создавать современный фронтенд с использованием Laravel и Livewire, многие разработчики предпочитают использовать возможности JavaScript-фреймворка, такого как Vue или React. Это позволяет разработчикам воспользоваться богатой экосистемой пакетов и инструментов JavaScript, доступных через NPM.

Однако, без дополнительных инструментов, совмещение Laravel с Vue или React потребовало бы решения различных сложных проблем, таких как маршрутизация на стороне клиента, гидратация данных и аутентификация. Маршрутизация на

стороне клиента часто упрощается с использованием специализированных фреймворков Vue / React, таких как [Nuxt](#) и [Next](#); однако гидратация данных и аутентификация остаются сложными и неудобными задачами при сочетании бэкенд-фреймворка, такого как Laravel, с этими фронтенд-фреймворками.

Кроме того, разработчикам приходится поддерживать два отдельных репозитория кода, часто требуется координировать обслуживание, релизы и развертывание в обоих репозиториях. Хотя эти проблемы не являются непреодолимыми, мы считаем, что это не продуктивный и приятный способ разработки приложений.

## Inertia

К счастью, Laravel предлагает лучшее из обоих миров. [Inertia](#) устраняет разрыв между вашим приложением Laravel и современным фронтеном на Vue или React. Это позволяет создавать полноценные, современные интерфейсы, используя Vue или React, при этом воспользовавшись маршрутами и контроллерами Laravel для решения задач по маршрутизации, гидратации данных и аутентификации — всё это в рамках единого репозитория кода. Таким образом, вы можете наслаждаться всей мощью Laravel и Vue/React, не сталкиваясь с ограничениями ни одного из этих инструментов.

После установки Inertia в вашем приложении Laravel, вы будете создавать маршруты и контроллеры так же, как обычно. Однако, вместо возвращения шаблона Blade из вашего контроллера, вы вернете страницу Inertia:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Inertia\Inertia;
use Inertia\Response;

class UserController extends Controller
{
    /**
     * Показать профиль для указанного пользователя
     */
    public function show(string $id): Response
    {
        return Inertia::render('Users/Profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

```
]);
}
}
```

Страница Inertia соответствует компоненту Vue или React, обычно сохраненному в каталоге `resources/js/Pages` вашего приложения. Данные, переданные странице с помощью метода `Inertia::render`, будут использоваться для гидратации “props” компонента страницы:

```
<script setup>
import Layout from '@/Layouts/Authenticated.vue';
import { Head } from '@inertiajs/vue3';

const props = defineProps(['user']);
</script>

<template>
    <Head title="User Profile" />

    <Layout>
        <template #header>
            <h2 class="font-semibold text-xl text-gray-800 leading-tight">
                Profile
            </h2>
        </template>

        <div class="py-12">
            Hello, {{ user.name }}
        </div>
    </Layout>
</template>
```

Как видите, Inertia позволяет использовать полную мощь Vue или React при создании фронтенда, обеспечивая легкий мост между вашим бэкендом, построенным на Laravel, и вашим фронтеном, работающим на JavaScript.

## Рендеринг на стороне сервера

Если вы беспокоитесь о том, чтобы приступить к использованию Inertia из-за необходимости рендеринга на стороне сервера в вашем приложении, не волнуйтесь. Inertia предоставляет [поддержку рендеринга на стороне сервера](#). И при развертывании вашего приложения через [Laravel Forge](#), легко обеспечить постоянную работу процесса серверного рендеринга Inertia.

# Стартовые наборы

Если вы хотите создать свой фронтенд, используя Inertia и Vue/React, вы можете воспользоваться нашими [стартовыми наборами](#) Breeze или Jetstream чтобы ускорить разработку вашего приложения. Оба этих стартовых комплекта генерируют структуру вашего бэкенда и фронтенда для аутентификации с использованием Vue / React, [Tailwind](#), и [Vite](#), чтобы вы могли просто приступить к реализации своей следующей большой идеи.

## # Сборка ресурсов (Bundling Assets)

Независимо от того, выберете ли вы разработку фронтенда с использованием Blade и Livewire или Vue/React и Inertia, вероятно, вам потребуется собрать CSS вашего приложения в готовые к продакшну ресурсы. Конечно же, если вы выберете построение фронтенда вашего приложения с использованием Vue или React, вам также потребуется собрать компоненты в готовые к использованию в браузере JavaScript-ресурсы.

По умолчанию Laravel использует [Vite](#) для сборки ваших ресурсов. Vite обеспечивает моментальную сборку и практически мгновенную замену модулей (HMR) во время локальной разработки. Во всех новых приложениях Laravel, включая те, которые используют наши [стартовые наборы](#), вы найдете файл `vite.config.js`, который загружает наш легкий плагин Laravel Vite, делая использование Vite приятным взаимодействием с приложениями Laravel.

Самый быстрый способ начать работу с Laravel и Vite – это начать разработку вашего приложения, используя [Laravel Breeze](#), нашего простейшего стартового набора, который ускоряет начало работы с вашим приложением, предоставляя основу для аутентификации на фронтенде и бэкенде.

Более подробную документацию по использованию Vite с Laravel можно найти в нашей [специализированной документации по сборке и компиляции ваших ресурсов..](#)

# Стартовые комплекты

## # Введение

## # Laravel Breeze

# Установка

# Breeze и Blade

# Breeze и Livewire

# Breeze & React / Vue

# Breeze & Next.js / API

## # Laravel Jetstream

## # Введение

Чтобы помочь вам быстрее начать работу с новым приложением Laravel, мы предлагаем стартовые комплекты для приложения, включая аутентификацию. Эти комплекты автоматически добавят в ваше приложение маршруты, контроллеры и шаблоны, необходимые для регистрации и авторизации пользователей.

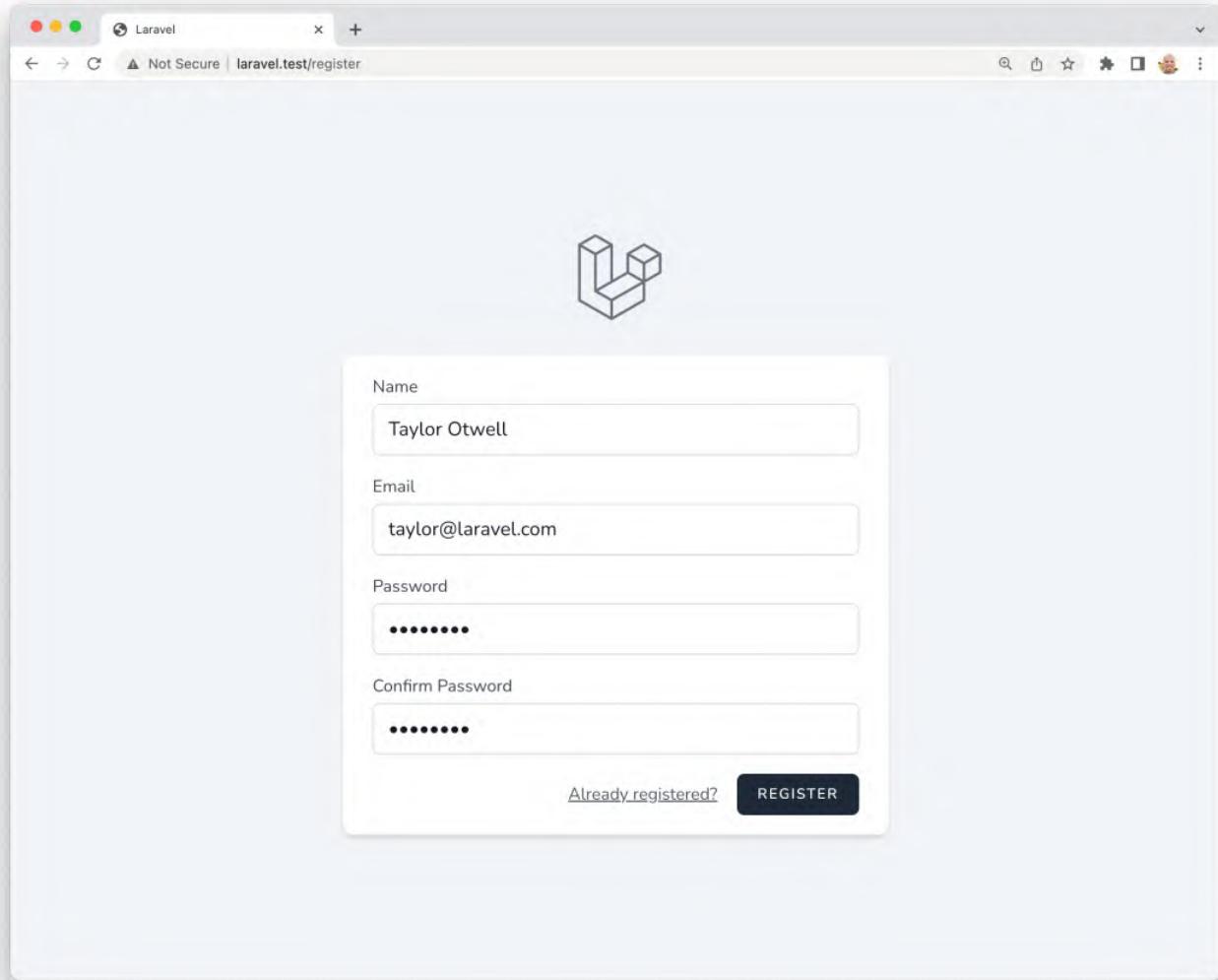
Использование стартовых комплектов необязательно — вы можете создать приложение с нуля, просто установив свежую копию Laravel. В любом случае, мы уверены, что у вас получится что-то замечательное!

## # Laravel Breeze

Laravel Breeze – это минимальная и простая реализация всех функций аутентификации Laravel, включая вход в систему, регистрацию, сброс пароля, подтверждение по электронной почте и подтверждение пароля. Кроме того, Breeze включает в себя простую “страницу профиля”, где пользователь может обновить свое имя, адрес электронной почты и пароль.

Представление по умолчанию в Laravel Breeze состоит из простых [шаблонов Blade](#), стилизованных с использованием [Tailwind CSS](#). Кроме того, Breeze предоставляет

опции для создания каркасов на основе [Livewire](#) или [Inertia](#), с выбором использования Vue или React для каркаса на основе Inertia.



Пример страницы регистрации Breeze

## Laravel Bootcamp

Если вы новичок в Laravel, не стесняйтесь присоединиться к [Laravel Bootcamp](#). Laravel Bootcamp научит вас создавать свое первое приложение на Laravel с использованием Breeze. Это отличный способ ознакомиться со всем, что Laravel и Breeze могут предложить.

## Установка

Во-первых, вам следует [создать новое приложение Laravel](#). Если вы создаете свое приложение с помощью [установщика Laravel](#), в процессе установки вас попросят установить Laravel Breeze. В противном случае вам придется следовать инструкциям по ручной установке ниже.

Если вы уже создали новое приложение Laravel без стартового набора, вы можете вручную установить Laravel Breeze с помощью Composer:

```
composer require laravel/breeze --dev
```

После установки пакета Laravel Breeze с помощью Composer, вы должны выполнить команду Artisan `breeze:install`. Эта команда публикует представления аутентификации, маршруты, контроллеры и другие ресурсы в вашем приложении. Laravel Breeze публикует всей свой код в вашем приложении, чтобы у вас была полная контроль и видимость над его функциями и реализацией.

Команда `breeze:install` запросит у вас предпочтительный стек фронтенда и фреймворк для тестирования:

```
php artisan breeze:install  
  
php artisan migrate  
npm install  
npm run dev
```

## Breeze и Blade

Стандартный стек Breeze – это Blade стек, который использует простые [шаблоны Blade](#) для отображения фронтенда вашего приложения. Вы можете установить Blade стек, вызвав команду `breeze:install` без дополнительных аргументов и выбрав Blade фронтенд стек. После установки структуры Breeze вам также следует скомпилировать фронтенд-ресурсы вашего приложения:

```
php artisan breeze:install  
  
php artisan migrate  
npm install  
npm run dev
```

Затем, вы можете перейти в своем веб-браузере по URL-адресам вашего приложения `/login` или `/register`. Все маршруты Breeze определены в файле `routes/auth.php`.

Чтобы узнать больше о компиляции CSS и JavaScript вашего приложения, ознакомьтесь с [документацией Laravel по Vite](#).

## Breeze и Livewire

Laravel Breeze также предлагает шаблоны для [Livewire](#). Livewire – это мощный способ создания динамических, реактивных пользовательских интерфейсов (UI) на фронтенде, используя только PHP.

Livewire отлично подходит для команд, которые в основном используют шаблоны Blade и ищут более простую альтернативу фреймворкам для создания SPA (Single Page Application) на JavaScript, таким как Vue и React.

Чтобы использовать стек Livewire, вы можете выбрать Livewire при выполнении команды Artisan `breeze:install`. После установки шаблонов Breeze вам следует запустить миграции базы данных:

```
php artisan breeze:install
```

```
php artisan migrate
```

## Breeze & React / Vue

Laravel Breeze также предоставляет средства сборки для React и Vue через фронтенд-реализацию [Inertia](#). Inertia позволяет создавать современные одностраничные приложения React и Vue с использованием классической маршрутизации на стороне сервера и контроллеров.

Inertia позволяет вам наслаждаться возможностями React и Vue на стороне фронтенда, объединенными с невероятной производительностью Laravel на стороне бэкенда и быстрой компиляцией [Vite](#). Чтобы использовать стек Inertia, вы можете выбрать стек Vue или React при выполнении команды Artisan `breeze:install`.

При выборе стека фронтенда Vue или React, установщик Breeze также предложит вам определить, хотите ли вы [Inertia SSR \(серверный рендеринг\)](#) или поддержку

TypeScript. После установки шаблонов Breeze вам также следует скомпилировать фронтенд-ресурсы вашего приложения:

```
php artisan breeze:install
```

```
php artisan migrate
npm install
npm run dev
```

Затем вы можете перейти по адресу `/login` или `/register` вашего приложения в веб-браузере. Все маршруты Breeze определены в файле `routes/auth.php`.

## Breeze & Next.js / API

Laravel Breeze также может создавать структуру аутентификационного API, готовую к аутентификации современных приложений на JavaScript, таких как [Next](#), [Nuxt](#) и другие. Для начала выберите стек API в качестве желаемого стека при выполнении команды Artisan `breeze:install`:

```
php artisan breeze:install
```

```
php artisan migrate
```

Во время установки Breeze добавит переменную окружения `FRONTEND_URL` в файл `.env` вашего приложения. Этот URL должен быть URL вашего JavaScript-приложения. Обычно это будет `http://localhost:3000` во время локальной разработки. Кроме того, убедитесь, что ваша переменная окружения `APP_URL` установлена в `http://localhost:8000`, который является URL по умолчанию для команды Artisan `serve`.

## Эталонная реализация Next.js

Наконец, вы готовы связать этот бэкэнд с выбранным вами интерфейсом. Следующая эталонная реализация интерфейса Breeze [доступна на GitHub](#). Этот интерфейс поддерживается Laravel и содержит тот же пользовательский интерфейс, что и традиционные стеки Blade и Inertia, предоставляемые Breeze.

## # Laravel Jetstream

В то время как Laravel Breeze обеспечивает простую и минимальную отправную точку для создания приложения Laravel, Jetstream дополняет эту функциональность более надежными функциями и дополнительными стеками технологий клиентского интерфейса. Для тех, кто новичок в Laravel, мы рекомендуем изучить основы работы с Laravel Breeze перед тем, как перейти на Laravel Jetstream.

Jetstream предоставляет красиво оформленный каркас приложения для Laravel и включает в себя функции входа, регистрации, проверки почты, двухфакторной аутентификации, управления сессиями, поддержку API с использованием Laravel Sanctum и опциональное управление командами. Jetstream разработан с использованием [Tailwind CSS](#) и предлагает выбор между фронтенд-структурами, основанными на [Livewire](#) или [Inertia](#).

Полное описание по установке Laravel Jetstream можно найти в [официальной документации Jetstream](#).

# Развертывание

- # Введение
- # Требования к серверу
- # Конфигурация сервера
  - # Nginx
  - # FrankenPHP
  - # Разрешения для папок
- # Оптимизация
  - # Кеширование конфигурации
  - # Кеширование событий
  - # Оптимизация загрузки маршрута
  - # Кеширование представлений
- # Режим отладки
- # Маршрут здоровья
- # Развертывание с помощью Forge / Vapor

## # Введение

Когда вы будете готовы развернуть свое приложение Laravel в эксплуатационном окружении, вы должны сделать несколько важных вещей, чтобы убедиться, что ваше приложение работает максимально эффективно. В этой документации мы рассмотрим несколько отличных отправных точек, чтобы убедиться, что ваше приложение Laravel развернуто правильно.

## # Требования к серверу

Фреймворк Laravel имеет несколько системных требований. Вы должны убедиться, что ваш веб-сервер имеет следующую минимальную версию PHP и расширения:

- PHP >= 8.2

- Расширение PHP Ctype
- Расширение PHP cURL
- Расширение PHP DOM
- Расширение PHP Fileinfo
- Расширение PHP Filter
- Расширение PHP Hash
- Расширение PHP Mbstring
- Расширение PHP OpenSSL
- Расширение PHP PCRE
- Расширение PHP PDO
- Расширение PHP Session
- Расширение PHP Tokenizer
- Расширение PHP XML

## # Конфигурация сервера

### Nginx

Если вы развертываете свое приложение на сервере, на котором работает Nginx, то вы можете использовать следующий конфигурационный файл в качестве отправной точки для настройки веб-сервера. Скорее всего, этот файл нужно будет настроить в зависимости от конфигурации вашего сервера. **Если вам нужна помощь в управлении вашим сервером, рассмотрите возможность использования собственной службы управления и развертывания серверов Laravel, такой как [Laravel Forge](#).**

Убедитесь, что, как и в конфигурации ниже, ваш веб-сервер направляет все запросы в файл `public/index.php` вашего приложения. Вы никогда не должны пытаться переместить файл `index.php` в корень вашего проекта, поскольку обслуживание приложения из корня проекта откроет доступ ко многим конфиденциальным файлам конфигурации из общедоступной сети Интернет:

```
server {
    listen 80;
    listen [::]:80;
    server_name example.com;
    root /srv/example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-Content-Type-Options "nosniff";

    index index.php;

    charset utf-8;

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

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
        fastcgi_hide_header X-Powered-By;
    }

    location ~ /.(?!well-known).* {
        deny all;
    }
}
```

## FrankenPHP

[FrankenPHP](#) также может использоваться для обслуживания ваших приложений Laravel. FrankenPHP — это современный сервер приложений PHP, написанный на Go. Чтобы обслуживать PHP-приложение Laravel с помощью FrankenPHP, вы можете просто вызвать его команду `php-server`:

```
frankenphp php-server -r public/
```

Чтобы воспользоваться более мощными функциями, поддерживаемыми FrankenPHP, такими как интеграция [Laravel Octane](#), HTTP/3, современное сжатие или возможность упаковывать приложения Laravel как автономные двоичные файлы, обратитесь к [документации Laravel](#) FrankenPHP.

## Разрешения для папок

Laravel потребуется разрешение на запись в каталоги `bootstrap/cache` и `storage`, поэтому вам следует убедиться, что у владельца процесса веб-сервера есть разрешение на запись в эти каталоги.

## # Оптимизация

При развертывании вашего приложения в рабочей среде необходимо кэшировать различные файлы, включая вашу конфигурацию, события, маршруты и представления. Laravel предоставляет единственную удобную команду Artisan [optimize](#), которая кэширует все эти файлы. Эту команду обычно следует вызывать как часть процесса развертывания вашего приложения:

```
php artisan optimize
```

Метод `optimize:clear` можно использовать для удаления всех файлов кэша, созданных командой [optimize](#), а также всех ключей в драйвере кэша по умолчанию:

```
php artisan optimize:clear
```

В следующей документации мы обсудим каждую из команд детальной оптимизации, выполняемых командой [optimize](#).

## Кэширование конфигурации

При развертывании вашего приложения в эксплуатационном окружении, вы должны убедиться, что вы выполнили команду `config:cache` Artisan в процессе развертывания:

```
php artisan config:cache
```

Эта команда объединит все файлы конфигурации Laravel в один кешированный файл, что значительно сократит количество обращений, которые фреймворк должен совершить к файловой системе при загрузке значений вашей конфигурации.

Если вы выполняете команду `config:cache` в процессе развертывания, вы должны быть уверены, что вызываете функцию `env` только из ваших файлов конфигурации. После кеширования конфигурации файл `.env` не будет загружаться, и все вызовы функции `env` для переменных файла `.env` вернут `null`.

## Кеширование событий

Вам следует кэшировать автоматически обнаруженное событие вашего приложения для сопоставления со слушателями во время процесса развертывания. Это можно сделать, вызвав команду Artisan `event:cache` во время развертывания:

```
php artisan event:cache
```

## Оптимизация загрузки маршрута

Если вы создаете большое приложение с множеством маршрутов, вам следует убедиться, что вы выполнили команду `route:cache` Artisan в процессе развертывания:

```
php artisan route:cache
```

Эта команда сокращает регистрацию всех маршрутов до одного вызова метода в кешированном файле, повышая производительность при регистрации сотен маршрутов.

## Кеширование представлений

При развертывании вашего приложения в эксплуатационном окружении, вы должны убедиться, что вы выполнили команду `view:cache` Artisan в процессе развертывания:

```
php artisan view:cache
```

Эта команда предварительно скомпилирует все ваши шаблоны Blade, чтобы они не компилировались во время запроса, повышая производительность каждого запроса, возвращающего шаблоном.

## # Режим отладки

Параметр отладки в файле конфигурации `config/app.php` определяет, сколько информации об ошибке фактически отображается пользователю. По умолчанию для этого параметра задано значение переменной среды `APP_DEBUG`, которая хранится в вашем файле `.env`.

**В вашем эксплуатационном окружении это значение всегда должно быть `false`. Если значение для переменной `APP_DEBUG` установлено как `true`, то вы рискуете раскрыть конфиденциальные значения конфигурации конечным пользователям вашего приложения.**

## # Маршрут здоровья

Laravel включает встроенный маршрут проверки работоспособности, который можно использовать для отслеживания статуса вашего приложения. В производственной среде этот маршрут можно использовать для сообщения о состоянии вашего приложения монитору работоспособности, балансировщику нагрузки или системе оркестрации, такой как Kubernetes.

По умолчанию маршрут проверки работоспособности обслуживается по адресу `/up` и возвращает HTTP-ответ 200, если приложение загрузилось без исключений. В

противном случае будет возвращен HTTP-ответ 500. Вы можете настроить URI для этого маршрута в файле `bootstrap/app` вашего приложения:

```
->withRouting(  
    web: __DIR__.'/../routes/web.php',  
    commands: __DIR__.'/../routes/console.php',  
    health: '/up', // [t1! удалить]  
    health: '/status', // [t1! добавить]  
)
```

When HTTP requests are made to this route, Laravel will also dispatch a `Illuminate\Foundation\Events\DiagnosingHealth` event, allowing you to perform additional health checks relevant to your application. Within a [listener](#) for this event, you may check your application's database or cache status. If you detect a problem with your application, you may simply throw an exception from the listener. Когда HTTP-запросы отправляются по этому маршруту, Laravel также отправляет событие `Illuminate\Foundation\Events\DiagnosingHealth`, позволяя вам выполнять дополнительные проверки работоспособности, относящиеся к вашему приложению. В [слушателе](#) для этого события вы можете проверить состояние базы данных или кэша вашего приложения. Если вы обнаружите проблему в своем приложении, вы можете просто выдать исключение из прослушивателя.

## # Развертывание с помощью Forge / Vapor

### Laravel Forge

Если вы не совсем готовы управлять конфигурацией своего собственного сервера или вам неудобно настраивать все различные службы, необходимые для запуска надежного приложения Laravel, то [Laravel Forge](#) – замечательная альтернатива.

Laravel Forge может создавать серверы на различных поставщиках инфраструктуры, таких как DigitalOcean, Linode, AWS и других. Кроме того, Forge устанавливает и управляет всеми инструментами, необходимыми для создания надежных приложений Laravel, таких как Nginx, MySQL, Redis, Memcached, Beanstalk и других.

Хотите полное руководство по развертыванию с использованием Laravel Forge? Проверьте [Laravel](#)

[Bootcamp](#) и [видео-серии Forge на Laracasts](#).

## Laravel Vapor

Если вам нужна полностью бессерверная платформа развертывания с автоматическим масштабированием, настроенная для Laravel, ознакомьтесь с [Laravel Vapor](#). Laravel Vapor – это платформа для бессерверного развертывания Laravel, работающая на AWS. Запустите свою инфраструктуру Laravel на Vapor и влюбитесь в масштабируемую простоту бессерверной архитектуры. Laravel Vapor настроен создателями Laravel для бесперебойной работы с фреймворком, поэтому вы можете продолжать писать свои приложения Laravel точно так, как вы привыкли.

# Жизненный цикл запроса

- # Введение
- # Обзор жизненного цикла
  - # Первые шаги
  - # HTTP-ядро и ядро консоли
  - # Маршрутизация
  - # Окончание
- # Сосредоточьтесь на сервис-провайдерах

## # Введение

При использовании любого инструмента в «реальном мире» вы чувствуете себя увереннее, если понимаете, как работает этот инструмент. Разработка приложений ничем не отличается. Вы чувствуете себя комфортнее и увереннее, когда понимаете, как работают используемые вами инструменты разработки.

Цель этого документа – дать вам хороший общий обзор того, как работает фреймворк Laravel. Если вы лучше узнаете общую структуру, то все станет менее «волшебным», и вы будете более уверены при создании своих приложений. Если вы не сразу поняли все термины, то не унывайте! Просто попытайтесь получить общее представление о том, что происходит, и ваши знания будут расти по мере изучения других разделов документации.

## # Обзор жизненного цикла

### Первые шаги

Точной входа для всех запросов к приложению Laravel является файл `public/index.php`. Все запросы направляются в этот файл конфигурацией вашего веб-сервера (Apache / Nginx). Файл `index.php` не содержит большого количества кода. Скорее, это отправная точка для загрузки остальной части фреймворка.

Файл `index.php` загружает автозагрузчик, созданный менеджером пакетов Composer, а затем извлекает экземпляр приложения Laravel из `bootstrap/app.php`. Первым действием, предпринимаемым самим Laravel, является создание экземпляра приложения / [контейнера служб](#).

## HTTP-ядро и ядро консоли

Затем входящий запрос отправляется либо HTTP-ядру, либо ядру консоли с использованием методов `handleRequest` или `handleCommand` экземпляра приложения, в зависимости от типа запроса, поступающего в приложение. Эти два ядра служат центральным местом, через которое проходят все запросы. А пока давайте сосредоточимся на ядре HTTP, которое является экземпляром `Illuminate\Foundation\Http\Kernel`.

HTTP-ядро определяет массив загрузчиков ([bootstrappers](#)), запускающихся до выполнения запроса. Эти загрузчики настраивают обработку ошибок, логирование, [определяют среду приложения](#) и выполняют другие задачи, которые необходимо выполнить до фактической обработки запроса. Обычно эти классы обращаются ко внутренней конфигурации Laravel, о которой вам не нужно беспокоиться.

Ядро HTTP также отвечает за передачу запроса через стек посредников. Эти посредники обрабатывают чтение и запись [сеанса HTTP](#), определяет, находится ли приложение в режиме обслуживания, [проверяет токен CSRF](#) и многое другое. Мы поговорим об этом подробнее в ближайшее время.

Сигнатура метода `handle` HTTP-ядра довольно проста: он получает запрос ([Request](#)) и возвращает ответ ([Response](#)). Думайте о ядре как о большом черном ящике, который представляет все ваше приложение. Дайте ему HTTP-запросы, и оно вернет HTTP-ответы.

## Сервис-провайдеры

Одно из самых важных действий инициализации ядра – это загрузка [сервис-провайдеров](#) для вашего приложения. Сервис-провайдеры отвечают за начальную настройку различных компонентов фреймворка, таких как база данных, очереди, валидация и маршрутизация.

Laravel пройдет по этому списку сервис-провайдеров и создаст экземпляры каждого из них. После создания экземпляров провайдеров, для каждого будет вызван метод `register`. Затем, как только все провайдеры будут зарегистрированы,

будет вызван метод `boot` каждого из них. Таким образом сервис-провайдеры могут расчитывать на то, что все связывания в контейнере служб будут зарегистрированы и доступны к моменту запуска метода `boot`.

Практически каждая основная функция, предлагаемая Laravel, инициализируется и настраивается с помощью сервис-провайдеров. Поскольку они инициализируют и настраивают множество функций, предоставляемых фреймворком, сервис-провайдеры являются самой важной частью всего процесса загрузки Laravel.

Хотя внутри фреймворка используются десятки сервис-провайдеров, у вас также есть возможность создать своих собственных. Вы можете найти список пользовательских или сторонних сервис-провайдеров, которые использует ваше приложение, в файле `bootstrap/providers.php`.

## Маршрутизация

После того как приложение было загружено и все поставщики служб зарегистрированы, `Request` будет передан маршрутизатору для исполнения. Маршрутизатор отправит запрос на маршрут или контроллер, а также запустит посредник для конкретного маршрута.

Посредники обеспечивают удобный механизм фильтрации или исследования HTTP-запросов, поступающих в ваше приложение. Например, Laravel содержит посредника, который проверяет аутентификацию пользователя вашего приложения. Если пользователь не аутентифицирован, посредник перенаправит пользователя, например, на экран входа в систему. Однако, если пользователь аутентифицирован, посредник позволит запросу продолжить работу в приложении. Некоторые посредники назначаются всем маршрутам в приложении, например `PreventRequestsDuringMaintenance`, тогда как некоторые назначаются только для определенных маршрутов или групп маршрутов. Вы можете узнать больше о посредниках, прочитав полную [документацию по посредникам](#).

Если запрос успешно проходит через всех посредников, назначенных определенному маршруту, то метод маршрута или контроллера будет выполнен, а ответ, возвращенный методом маршрута или контроллера, будет отправлен обратно через цепочку посредников маршрута.

## Окончание

Когда метод маршрута или контроллера вернет ответ, тогда ответ отправится обратно через посредников маршрута, обеспечивая приложению возможность изменения или проверки исходящего ответа.

Наконец, как только ответ проходит через посредников, метод `handle` ядра HTTP возвращает объект ответа в `handleRequest` экземпляра приложения, и этот метод вызывает метод `send` для возвращенного ответа. Метод `send` отправляет содержимое ответа в веб-браузер пользователя. Мы завершили весь жизненный цикл запроса Laravel!

## # Сосредоточьтесь на сервис-провайдерах

Сервис-провайдеры действительно являются ключом к начальной загрузке приложения Laravel. Экземпляр приложения создается, сервис-провайдеры регистрируются, и запрос передается загруженному приложению. Это действительно так просто!

Очень важно иметь четкое представление о том, как создается и загружается приложение Laravel через сервис-провайдеры. Пользовательские сервис-провайдеры для вашего приложения хранятся в каталоге `app/Providers`.

По умолчанию провайдер `AppServiceProvider` относительно пуст. Этот провайдер является отличным местом для добавления собственной инициализации и связываний контейнера служб вашего приложения. Для больших приложений вы можете создать несколько поставщиков, каждый из которых детализирует начальную загрузку для конкретных сервисов, используемых вашим приложением.

# Контейнер служб (service container)

## # Введение

# Неконфигурируемое внедрение

# Когда использовать контейнер

## # Связывание

# Основы связываний

# Связывание интерфейсов и реализаций

# Контекстная привязка

# Контекстуальные атрибуты

# Связывание примитивов

# Связывание типизированных вариаций

# Добавление меток

# Расширяемость связываний

## # Извлечение

# Метод make

# Автоматическое внедрение зависимостей

## # Вызов и внедрение метода

## # События контейнера

# Перепривязка

## # PSR-11

## # Введение

Контейнер служб (service container, сервис-контейнер) Laravel – это мощный инструмент для управления зависимостями классов и выполнения внедрения зависимостей. Внедрение зависимостей – это причудливая фраза, которая по существу означает следующее: зависимости классов «вводятся» в класс через конструктор в виде аргументов или, в некоторых случаях, через методы-сеттеры. При создании класса или вызове методов фреймворк смотрит на список аргументов и, если нужно, создаёт экземпляры необходимых классов и сам подаёт их на вход конструктора или метода.

Давайте посмотрим на простой пример:

```
<?php

namespace App\Http\Controllers;

use App\Services\AppleMusic;
use Illuminate\View\View;

class PodcastController extends Controller
{
    /**
     * Создать новый экземпляр контроллера.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(
        protected AppleMusic $apple,
    ) {}

    /**
     * Показать информацию о данном подкасте.
     */
    public function show(string $id): View
    {
        return view('podcasts.show', [
            'podcast' => $this->apple->findPodcast($id)
        ]);
    }
}
```

В этом примере `PodcastController` необходимо получить подкасты из источника данных, такого как Apple Music. Итак, мы **внедрим** сервис, способный извлекать подкасты. Поскольку служба внедрена, мы можем легко «имитировать» или создать фиктивную реализацию службы `AppleMusic` при тестировании нашего приложения.

Глубокое понимание контейнера служб Laravel необходимо для создания большого, мощного приложения, а также для внесения вклада в само ядро Laravel.

## Неконфигурируемое внедрение

Если класс не имеет зависимостей или зависит только от других конкретных классов (не интерфейсов), контейнер не нужно инструментировать о том, как создавать этот класс. Например, вы можете поместить следующий код в свой файл `routes/web.php`:

```
<?php

class Service
{
    // ...

}

Route::get('/', function (Service $service) {
    die($service::class);
});
```

В этом примере, при посещении `/` вашего приложения, маршрут автоматически получит класс `Service` и внедрит его в обработчике вашего маршрута. Это меняет правила игры. Это означает, что вы можете разработать свое приложение и воспользоваться преимуществами внедрения зависимостей, не беспокоясь о раздутых файлах конфигурации.

К счастью, многие классы, которые вы будете писать при создании приложения Laravel, автоматически получают свои зависимости через контейнер, включая [контроллеры](#), [слушатели событий](#), [посредники](#) и т.д. Кроме того, вы можете указать зависимости в методе `handle` обработки [заданий в очереди](#). Как только вы почувствуете всю мощь автоматического неконфигурируемого внедрения зависимостей, вы почувствуете невозможность разработки без нее.

## Когда использовать контейнер

Благодаря неконфигурируемому внедрению, вы часто будете объявлять типы зависимостей в маршрутах, контроллерах, слушателях событий и других местах, не взаимодействуя с контейнером напрямую. Например, вы можете указать объект `Illuminate\Http\Request` в определении вашего маршрута, для того, чтобы легко получить доступ к текущему запросу. Несмотря на то, что нам никогда не нужно взаимодействовать с контейнером для написания этого кода, он управляет внедрением этих зависимостей за кулисами:

```
use Illuminate\Http\Request;
```

```
Route::get('/', function (Request $request) {
    // ...
});
```

Во многих случаях, благодаря автоматическому внедрению зависимостей и [фасадам](#), вы можете строить приложения Laravel без необходимости **когда-либо** вручную связывать или извлекать что-либо из контейнера. **В каких же случаях есть необходимость вручную взаимодействовать с контейнером?**. Давайте рассмотрим две ситуации.

Во-первых, если вы пишете класс, реализующий интерфейс, и хотите объявить тип этого интерфейса в конструкторе маршрута или класса, то вы должны [сообщить контейнеру, как получить этот интерфейс](#). Во-вторых, если вы [пишете пакет Laravel](#), которым планируете поделиться с другими разработчиками Laravel, вам может потребоваться связать службы вашего пакета в контейнере.

## # Связывание

### Основы связываний

#### Простое связывание

Почти все ваши связывания в контейнере служб будут зарегистрированы в [поставщиках служб](#), поэтому в большинстве этих примеров будет продемонстрировано использование контейнера в этом контексте.

Внутри поставщика служб у вас всегда есть доступ к контейнеру через свойство `$this->app`. Мы можем зарегистрировать связывание, используя метод `bind`, передав имя класса или интерфейса, которые мы хотим зарегистрировать, вместе с замыканием, возвращающим экземпляр класса:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Обратите внимание, что мы получаем сам контейнер в качестве аргумента. Затем мы можем использовать контейнер для извлечения под-зависимостей объекта, который мы создаем.

Как уже упоминалось, вы обычно будете взаимодействовать с контейнером внутри поставщиков служб; однако, если вы хотите взаимодействовать с контейнером в других частях приложения, вы можете сделать это через [фасад App](#):

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

Вы можете использовать метод [bindIf](#) для регистрации привязки контейнера только в том случае, если привязка уже не была зарегистрирована для данного типа:

```
$this->app->bindIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Нет необходимости привязывать классы в контейнере, если они не зависят от каких-либо интерфейсов. Контейнеру не нужно указывать, как создавать эти объекты, поскольку он может автоматически извлекать эти объекты с помощью рефлексии.

## Связывание одиночек

Метод [singleton](#) связывает в контейнере класс или интерфейс, который должен быть извлечен только один раз. При последующих обращениях к этому классу из контейнера будет возвращен полученный ранее экземпляр объекта:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->singleton(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Вы можете использовать метод `singletonIf` для регистрации синглтон-привязки контейнера только в том случае, если привязка уже не была зарегистрирована для данного типа:

```
$this->app->singletonIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

## Связывание одиночек с заданной областью действия

Метод `scoped` связывает в контейнере класс или интерфейс, который должен быть извлечен только один раз в течение данного жизненного цикла запроса / задания Laravel. Хотя этот метод похож на метод `singleton` похож на метод `scoped`, экземпляры, зарегистрированные с помощью метода `scoped`, будут сбрасываться всякий раз, когда приложение Laravel запускает новый «жизненный цикл», например, когда [Laravel Octane](#) обрабатывает новый запрос или когда [очереди](#) обрабатывают новое задание:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Вы можете использовать метод `scopedIf` для регистрации привязки контейнера с ограниченной областью действия, только если привязка еще не зарегистрирована для данного типа:

```
$this->app->scopedIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

```
});
```

## Связывание экземпляров

Вы также можете привязать существующий экземпляр объекта в контейнере, используя метод `instance`. Переданный экземпляр всегда будет возвращен из контейнера при последующих вызовах:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);
```

## Связывание интерфейсов и реализаций

Очень мощная функция контейнера служб – это его способность связывать интерфейс с конкретной реализацией. Например, предположим, что у нас есть интерфейс `EventPusher` и реализация `RedisEventPusher`. После того как мы написали нашу реализацию `RedisEventPusher` этого интерфейса, мы можем зарегистрировать его в контейнере следующим образом:

```
use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);
```

Эта запись сообщает контейнеру, что он должен внедрить `RedisEventPusher`, когда классу требуется реализация `EventPusher`. Теперь мы можем указать интерфейс `EventPusher` в конструкторе класса, который будет извлечен контейнером. Помните, что контроллеры, слушатели событий, посредники и некоторые другие типы классов в приложениях Laravel всегда выполняются с помощью контейнера:

```
use App\Contracts\EventPusher;

/**
 * Создать новый экземпляр класса.
 */
public function __construct()
```

```
protected EventPusher $pusher,  
) {}
```

## Контекстная привязка

Иногда у вас может быть два класса, которые используют один и тот же интерфейс, но вы хотите внедрить разные реализации в каждый класс. Например, два контроллера могут зависеть от разных реализаций [контракта Illuminate\Contracts\Filesystem\Filesystem](#). Laravel предлагает простой и понятный интерфейс для определения этого поведения:

```
use App\Http\Controllers\PhotoController;  
use App\Http\Controllers\UploadController;  
use App\Http\Controllers\VideoController;  
use Illuminate\Contracts\Filesystem\Filesystem;  
use Illuminate\Support\Facades\Storage;  
  
$this->app->when(PhotoController::class)  
    ->needs(Filesystem::class)  
    ->give(function () {  
        return Storage::disk('local');  
    });  
  
$this->app->when([VideoController::class, UploadController::class])  
    ->needs(Filesystem::class)  
    ->give(function () {  
        return Storage::disk('s3');  
    });
```

## Контекстуальные атрибуты

Поскольку контекстная привязка часто используется для внедрения реализаций драйверов или значений конфигурации, Laravel предлагает множество атрибутов контекстной привязки, которые позволяют внедрять эти типы значений без ручного определения контекстных привязок у ваших поставщиков услуг.

Например, атрибут `Storage` может использоваться для внедрения определенного [диска хранения](#):

```
<?php
```

```
namespace App\Http\Controllers;
```

```
use Illuminate\Container\Attributes\Storage;
use Illuminate\Contracts\Filesystem\Filesystem;

class PhotoController extends Controller
{
    public function __construct(
        #[Storage('local')] protected Filesystem $filesystem
    )
    {
        // ...
    }
}
```

В дополнение к атрибуту `Storage`, Laravel предлагает атрибуты `Auth`, `Cache`, `Config`, `DB`, `Log`, `RouteParameter` и `Tag`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Container\Attributes\Auth;
use Illuminate\Container\Attributes\Cache;
use Illuminate\Container\Attributes\Config;
use Illuminate\Container\Attributes\DB;
use Illuminate\Container\Attributes\Log;
use Illuminate\Container\Attributes\Tag;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Contracts\Database\Connection;
use Psr\Log\LoggerInterface;

class PhotoController extends Controller
{
    public function __construct(
        #[Auth('web')] protected Guard $auth,
        #[Cache('redis')] protected Repository $cache,
        #[Config('app.timezone')] protected string $timezone,
        #[DB('mysql')] protected Connection $connection,
        #[Log('daily')] protected LoggerInterface $log,
        #[Tag('reports')] protected iterable $reports,
    )
    {
        // ...
    }
}
```

Кроме того, Laravel предоставляет атрибут `CurrentUser` для добавления текущего аутентифицированного пользователя в заданный маршрут или класс:

```
use App\Models\User;
use Illuminate\Container\Attributes\CurrentUser;

Route::get('/user', function (#[CurrentUser] User $user) {
    return $user;
})->middleware('auth');
```

## Определение пользовательских атрибутов

Вы можете создавать свои собственные контекстные атрибуты, реализуя контракт `Illuminate\Contracts\Container\ContextualAttribute`. Контейнер вызовет метод `resolve` вашего атрибута, который должен разрешить значение, которое должно быть введено в класс, использующий атрибут. В приведенном ниже примере мы повторно реализуем встроенный атрибут Laravel `Config`:

```
<?php

namespace App\Attributes;

use Illuminate\Contracts\Container\ContextualAttribute;

#[Attribute(Attribute::TARGET_PARAMETER)]
class Config implements ContextualAttribute
{
    /**
     * Create a new attribute instance.
     */
    public function __construct(public string $key, public mixed $default = null)
    {
    }

    /**
     * Resolve the configuration value.
     *
     * @param self $attribute
     * @param \Illuminate\Contracts\Container\Container $container
     * @return mixed
     */
    public static function resolve(self $attribute, Container $container)
    {
        return $container->make('config')->get($attribute->key, $attribute->default);
    }
}
```

```
    }  
}
```

## Связывание примитивов

Иногда у вас может быть класс, который получает некоторые внедренные классы, но также нуждается в примитиве, таком как целое число. Вы можете легко использовать контекстную привязку, чтобы внедрить любое значение, которое может понадобиться вашему классу:

```
use App\Http\Controllers\UserController;  
  
$this->app->when(UserController::class)  
    ->needs('$variableName')  
    ->give($value);
```

Иногда класс может зависеть от массива экземпляров, объединенных [меткой](#). Используя метод `giveTagged`, вы можете легко их внедрить:

```
$this->app->when(ReportAggregator::class)  
    ->needs('$reports')  
    ->giveTagged('reports');
```

Если вам нужно внедрить значение из одного из конфигурационных файлов вашего приложения, то вы можете использовать метод `giveConfig`:

```
$this->app->when(ReportAggregator::class)  
    ->needs('$timezone')  
    ->giveConfig('app.timezone');
```

## Связывание типизированных вариаций

Иногда у вас может быть класс, который получает массив типизированных объектов с использованием переменного количества аргументов (*прим. перев.: далее «вариации»*) конструктора:

```
<?php
```

```
use App\Models\Filter;
use App\Services\Logger;

class Firewall
{
    /**
     * Создать новый экземпляр класса.
     */
    public function __construct(
        protected Logger $logger,
        Filter ...$filters,
    ) {
        $this->filters = $filters;
    }
}
```

Используя контекстную привязку, вы можете внедрить такую зависимость, используя метод `give` с замыканием, которое возвращает массив внедряемых экземпляров `Filter`:

```
$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give(function (Application $app) {
        return [
            $app->make(NullFilter::class),
            $app->make(ProfanityFilter::class),
            $app->make(TooLongFilter::class),
        ];
});
```

Для удобства вы также можете просто передать массив имен классов, которые будут предоставлены контейнером всякий раз, когда для `Firewall` нужны экземпляры `Filter`:

```
$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give([
        NullFilter::class,
        ProfanityFilter::class,
        TooLongFilter::class,
    ]);
```

## Метки вариативных зависимостей

Иногда класс может иметь вариативную зависимость, указывающую на тип как переданный класс (`Report ...$reports`). Используя методы `needs` и `giveTagged`, вы можете легко внедрить все привязки контейнера с этой [меткой](#) для указанной зависимости:

```
$this->app->when(ReportAggregator::class)
    ->needs(Report::class)
    ->giveTagged('reports');
```

## Добавление меток

Иногда может потребоваться получить все привязки определенной «категории». Например, возможно, вы создаете анализатор отчетов, который получает массив из множества различных реализаций интерфейса `Report`. После регистрации реализаций `Report` вы можете назначить им метку с помощью метода `tag`:

```
$this->app->bind(CpuReport::class, function () {
    // ...
});

$this->app->bind(MemoryReport::class, function () {
    // ...
});

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

После того как службы помечены, вы можете легко все их получить с помощью метода `tagged`:

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

## Расширяемость связываний

Метод `extend` позволяет модифицировать извлеченные службы. Например, когда служба получена, вы можете выполнить дополнительный код для декорирования

или конфигурирования службы. Метод `extend` принимает два аргумента: класс службы, который вы расширяете, и замыкание, которое должно возвращать модифицированную службу. Замыкание получает службу, которая извлечения, и экземпляр контейнера:

```
$this->app->extend(Service::class, function (Service $service, Application $app) {
    return new DecoratedService($service);
});
```

## # Извлечение

### Метод `make`

Вы можете использовать метод `make` для извлечения экземпляра класса из контейнера. Метод `make` принимает имя класса или интерфейса, который вы хотите получить:

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

Если некоторые зависимости вашего класса не могут быть разрешены через контейнер, вы можете ввести их, передав их как ассоциативный массив в метод `makeWith`. Например, мы можем вручную передать конструктору аргумент `$id`, требуемый службой `Transistor`:

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

Метод `bound` может быть использован для определения, был ли класс или интерфейс явно привязан в контейнере:

```
if ($this->app->bound(Transistor::class)) {
    // ...
}
```

Если вы находитесь за пределами поставщика служб и не имеете доступа к переменной `$app`, вы можете использовать [фасад App](#) для получения экземпляра класса из контейнера:

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);
```

Если вы хотите, чтобы сам экземпляр контейнера Laravel был внедрен в класс, извлекаемый контейнером, вы можете указать класс [Illuminate\Container\Container](#) в конструкторе вашего класса:

```
use Illuminate\Container\Container;

/**
 * Создать новый экземпляр класса.
 */
public function __construct(
    protected Container $container,
) {}
```

## Автоматическое внедрение зависимостей

Важно, что в качестве альтернативы, вы можете объявить тип зависимости в конструкторе класса, который извлекается контейнером, включая [контроллеры](#), [слушатели событий](#), [посредники](#) и т.д. Кроме того, вы можете объявить зависимости в методе `handle` обработки [заданий в очереди](#). На практике именно так контейнер должен извлекать большинство ваших объектов.

Например, вы можете объявить сервис, определенный вашим приложением, в конструкторе контроллера. Сервис будет автоматически получен и внедрен в класс:

```
<?php

namespace App\Http\Controllers;

use App\Services\AppleMusic;

class PodcastController extends Controller
```

```
{  
    /**  
     * Создать новый экземпляр контроллера.  
     */  
    public function __construct(  
        protected AppleMusic $apple,  
    ) {}  
  
    /**  
     * Показать информацию о данном подкасте.  
     */  
    public function show(string $id): Podcast  
    {  
        return $this->apple->findPodcast($id);  
    }  
}
```

## # Вызов и внедрение метода

Иногда вам может потребоваться вызвать метод для экземпляра объекта, позволяя контейнеру автоматически вводить зависимости этого метода. Например, учитывая следующий класс:

```
<?php  
  
namespace App;  
  
use App\Services\AppleMusic;  
  
class PodcastStats  
{  
    /**  
     * Generate a new podcast stats report.  
     */  
    public function generate(AppleMusic $apple): array  
    {  
        return [  
            // ...  
        ];  
    }  
}
```

Вы можете вызвать метод `generate` через контейнер следующим образом:

```
use App\PodcastStats;
use Illuminate\Support\Facades\App;

$stats = App::call([new PodcastStats, 'generate']);
```

Метод `call` принимает любой вызываемый PHP-код. Метод контейнера `call` может даже использоваться для вызова замыкания при автоматическом внедрении его зависимостей:

```
use App\Services\AppleMusic;
use Illuminate\Support\Facades\App;

$result = App::call(function (AppleMusic $apple) {
    // ...
});
```

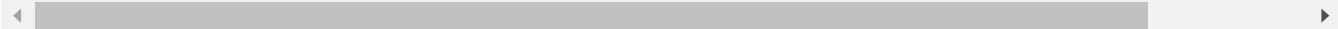
## # События контейнера

Контейнер служб инициирует событие каждый раз, когда извлекает объект. Вы можете прослушать это событие с помощью метода `resolving`:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor, Application $app) {
    // Вызывается, когда контейнер извлекает объекты типа `Transistor` ...
});

$this->app->resolving(function (mixed $object, Application $app) {
    // Вызывается, когда контейнер извлекает объект любого типа ...
});
```



Как видите, извлекаемый объект будет передан в замыкание, что позволит вам установить любые дополнительные свойства объекта до того, как он будет передан его получателю.

## Перепривязка

Метод `rebinding` позволяет вам прослушивать, когда служба повторно привязывается к контейнеру, то есть она снова регистрируется или переопределется после первоначальной привязки. Это может быть полезно, когда вам нужно обновить зависимости или изменить поведение каждый раз при обновлении определенной привязки:

```
use App\Contracts\PodcastPublisher;
use App\Services\SpotifyPublisher;
use App\Services\TransistorPublisher;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(PodcastPublisher::class, SpotifyPublisher::class);

$this->app->rebinding(
    PodcastPublisher::class,
    function (Application $app, PodcastPublisher $newInstance) {
        //
    },
);

// New binding will trigger rebinding closure...
$this->app->bind(PodcastPublisher::class, TransistorPublisher::class);
```

## # PSR-11

Контейнер служб Laravel реализует интерфейс [PSR-11](#). Поэтому вы можете объявить тип интерфейса контейнера PSR-11, чтобы получить экземпляр контейнера Laravel:

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    // ...
});
```

Иключение выбрасывается, если данный идентификатор не может быть получен. Исключением будет экземпляр [Psr\Container\NotFoundExceptionInterface](#), если идентификатор никогда не был привязан. Если идентификатор был

привязан, но не может быть извлечен, будет брошен экземпляр `Psr\Container\ContainerExceptionInterface`.

# Сервис-провайдеры

- # Введение
- # Написание сервис-провайдеров
  - # Метод register
  - # Метод boot
- # Регистрация сервис-провайдеров
- # Отложенные сервис-провайдеры

## # Введение

Сервис-провайдеры – это центральное место начальной загрузки всех приложений Laravel. Ваше собственное приложение, а также все основные службы и сервисы Laravel загружаются через них.

Но, что мы подразумеваем под «начальной загрузкой»? В общем, мы имеем в виду **регистрацию** элементов, включая регистрацию связываний контейнера служб (service container), слушателей событий (event listener), посредников (middleware) и даже маршрутов (route). Сервис-провайдеры являются центральным местом для конфигурирования приложения.

Laravel использует десятки поставщиков услуг внутренне для инициализации своих основных сервисов, таких как почтовый сервис, очереди, кэш и другие. Многие из этих поставщиков являются “отложенными”, что означает, что они не будут загружены при каждом запросе, а только когда нужны фактические сервисы, которые они предоставляют.

Все определенные пользователем поставщики услуг регистрируются в файле `bootstrap/providers.php`. В этой документации вы узнаете, как писать собственные сервис-провайдеры и зарегистрировать их в приложении Laravel.

Если вы хотите узнать больше о том, как Laravel обрабатывает запросы и работает изнутри,

ознакомьтесь с нашей документацией по [жизненному циклу запроса Laravel](#).

## # Написание сервис-провайдеров

Все сервис-провайдеры расширяют класс `Illuminate\Support\ServiceProvider`. Большинство сервис-провайдеров содержат метод `register` и `boot`. В рамках метода `register` следует **только связывать (bind) сущности в контейнере служб**. Никогда не следует пытаться зарегистрировать каких-либо слушателей событий, маршруты или что-то другое в методе `register`.

Чтобы сгенерировать новый сервис-провайдер, используйте команду `make:provider Artisan`. Laravel автоматически зарегистрирует вашего нового сервис-провайдера в файле `bootstrap/providers.php` вашего приложения:

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

### Метод register

Как упоминалось ранее, в рамках метода `register` следует только связывать сущности в [контейнере служб](#). Никогда не следует пытаться зарегистрировать слушателей событий, маршруты или что-то другое в методе `register`. В противном случае вы можете случайно воспользоваться подсистемой, чей сервис-провайдер еще не загружен.

Давайте взглянем на рядовой сервис-провайдер приложения. В любом из методов сервис-провайдера у вас всегда есть доступ к свойству `$app`, которое обеспечивает доступ к контейнеру служб:

```
<?php  
  
namespace App\Providers;  
  
use App\Services\Riak\Connection;  
use Illuminate\Contracts\Foundation\Application;  
use Illuminate\Support\ServiceProvider;  
  
class RiakServiceProvider extends ServiceProvider
```

```
{  
    /**  
     * Регистрация любых служб приложения.  
     */  
    public function register(): void  
    {  
        $this->app->singleton(Connection::class, function (Application $app) {  
            return new Connection(config('riak'));  
        });  
    }  
}
```

Этот сервис-провайдер определяет только метод `register` и использует этот метод для указания, какая именно реализация `App\Services\Riak\Connection` будет применена в нашем приложении – при помощи контейнера служб. Если вы еще не знакомы с контейнером служб Laravel, ознакомьтесь с [его документацией](#).

## Свойства `bindings` и `singletons`

Если ваш сервис-провайдер регистрирует много простых связываний, вы можете использовать свойства `bindings` и `singletons` вместо ручной регистрации каждого связывания контейнера. Когда сервис-провайдер загружается фреймворком, он автоматически проверяет эти свойства и регистрирует их связывания:

```
<?php  
  
namespace App\Providers;  
  
use App\Contracts\DowntimeNotifier;  
use App\Contracts\ServiceProvider;  
use App\Services\DigitalOceanServiceProvider;  
use App\Services\PingdomDowntimeNotifier;  
use App\Services\ServerToolsProvider;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Все связывания контейнера, которые должны быть зарегистрированы.  
     *  
     * @var array  
     */  
    public $bindings = [  
        ServerProvider::class => DigitalOceanServiceProvider::class,  
    ];
```

```

/**
 * Все синглтоны контейнера, которые должны быть зарегистрированы.
 *
 * @var array
 */
public $singletons = [
    DowntimeNotifier::class => PingdomDowntimeNotifier::class,
    ServerProvider::class => ServerToolsProvider::class,
];
}

```

## Метод boot

Итак, что, если нам нужно зарегистрировать [компоновщик шаблонов](#) в нашем сервис-провайдере? Это должно быть сделано в рамках метода `boot`. **Этот метод вызывается после регистрации всех остальных сервис-провайдеров**, что означает, что в этом месте у вас уже есть доступ ко всем другим службам, которые были зарегистрированы фреймворком:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        View::composer('view', function () {
            // ...
        });
    }
}

```

## Внедрение зависимости в методе boot

Вы можете указывать тип зависимостей в методе `boot` сервис-провайдера. [Контейнер служб](#) автоматически внедрит любые необходимые зависимости:

```
use Illuminate\Contracts\Routing\ResponseFactory;

/**
 * Загрузка любых служб приложения.
 */
public function boot(ResponseFactory $response): void
{
    $response->macro('serialized', function (mixed $value) {
        // ...
    });
}
```

## # Регистрация сервис-провайдеров

Все поставщики услуг регистрируются в файле конфигурации `bootstrap/providers.php`. Этот файл возвращает массив, который содержит имена классов поставщиков услуг вашего приложения:

```
<?php

return [
    App\Providers\AppServiceProvider::class,
];
```

Когда вы вызываете команду `make:provider` в Artisan, Laravel автоматически добавит сгенерированный провайдер в файл `bootstrap/providers.php`. Однако, если вы создали класс провайдера вручную, вы должны вручную добавить класс провайдера в массив:

```
<?php

return [
    App\Providers\AppServiceProvider::class,
    App\Providers\ComposerServiceProvider::class, // Ваш новый провайдер
];
```

## # Отложенные сервис-провайдеры

Если ваш сервис-провайдер регистрирует **только** связывания в [контейнере служб](#), вы можете отложить его регистрацию до тех пор, пока одно из зарегистрированных связываний не понадобится. Отсрочка загрузки такого сервис-провайдера повысит производительность вашего приложения, так как он не загружается из файловой системы при каждом запросе.

Laravel составляет и сохраняет список всех служб, предоставляемых отложенными сервис-провайдерами, а также имя класса сервис-провайдера. Laravel загрузит сервис-провайдер только при необходимости в одной из этих служб.

Чтобы отложить загрузку сервис-провайдера, реализуйте интерфейс [\Illuminate\Contracts\Support\DeferrableProvider](#), описав метод `provides`. Метод `provides` должен вернуть связывания контейнера службы, регистрируемые данным классом:

```
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Contracts\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Получить службы, предоставляемые поставщиком.
     *
     * @return array<int, string>
     */
    public function provides(): array
    {
        return [Connection::class];
```

}  
 }

# Фасады (Facades)

# Введение

# Когда использовать фасады

# Фасады против внедрения зависимостей

# Фасады против глобальных помощников

# Как фасады работают

# Фасады в реальном времени

# Справочник фасадов

## # Введение

В документации Laravel вы увидите примеры кода, демонстрирующего взаимодействия с функционалом Laravel через «фасады». Фасады предоставляют «статический» интерфейс для классов, доступных в [контейнере служб](#) приложения. Laravel из коробки включает множество фасадов, обеспечивающих доступ почти ко всему функционалу Laravel.

Фасады Laravel служат «статическими прокси» для базовых классов в контейнере служб, обеспечивая преимущества краткого, выразительного синтаксиса при сохранении большей тестируемости и гибкости, чем традиционные статические методы. Если вы не совсем понимаете, как фасады работают под капотом просто идите вперед – оставьте этот момент на будущее, примите к сведению синтаксис фасадов и то, что они похожи на статические вызовы, и просто продолжайте изучать Laravel.

Все фасады Laravel определены в пространстве имён [Illuminate\Support\Facades](#). Таким образом, мы можем легко получить доступ к такому фасаду:

```
use Illuminate\Support\Facades\Cache;  
use Illuminate\Support\Facades\Route;
```

```
Route::get('/cache', function () {
```

```
    return Cache::get('key');
});
```

В документации Laravel во многих примерах будут использоваться фасады для демонстрации различного функционала фреймворка.

## Глобальные помощники

В дополнении к фасадам, Laravel предлагает множество глобальных «вспомогательных функций», которые упрощают взаимодействие с общими функциями Laravel. Вот некоторые из глобальных помощников, с которыми вы можете взаимодействовать – это `view`, `response`, `url`, `config` и т.д. Каждый помощник, предлагаемый Laravel, задокументирован с соответствующей функцией; однако полный список доступен в специальной [документации глобальных помощников](#).

Например, вместо использования фасада `Illuminate\Support\Facades\Response` для генерации ответа JSON, мы можем просто использовать функцию `response`. Поскольку помощники доступны глобально, то вам не нужно импортировать какие-либо классы, чтобы использовать их:

```
use Illuminate\Support\Facades\Response;

Route::get('/users', function () {
    return Response::json([
        // ...
    ]);
});

Route::get('/users', function () {
    return response()->json([
        // ...
    ]);
});
```

## # Когда использовать фасады

У фасадов много преимуществ. Они предоставляют краткий, запоминающийся синтаксис, позволяющий вам использовать функции Laravel, не запоминая длинные имена классов, которые необходимо вводить или конфигурировать вручную.

Более того, благодаря уникальному использованию динамических методов PHP их легко протестировать.

Однако при использовании фасадов необходимо соблюдать некоторую осторожность. Основная опасность фасадов – «разрастание» класса. Поскольку фасады настолько просты в использовании и не требуют внедрений, что легко оказывается на разрастании класса и использовании множества фасадов в одном классе. При использовании внедрения зависимостей этот потенциал снижается за счет визуальной обратной связи, которую дает большой конструктор, сигнализируя о том, что ваш класс становится слишком большим. Поэтому, используя фасады, обратите особое внимание на размер вашего класса, чтобы уровень его ответственности оставался узким. Если ваш класс становится слишком большим, рассмотрите возможность разделения его на несколько более мелких классов.

## Фасады против внедрения зависимостей

Одним из основных преимуществ внедрения зависимостей является возможность изменения реализации внедренного класса. Это полезно во время тестирования, так как вы можете вставить имитацию или заглушку и утверждать, что для заглушки были вызваны различные методы.

Как правило, невозможно имитировать или заглушить действительно статический метод класса. Однако, поскольку фасады используют динамические методы для проксирования вызовов методов к объектам, извлекаемым из контейнера служб, мы фактически можем тестировать фасады так же, как тестировали бы внедренный экземпляр класса. Например, учитывая следующий маршрут:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Используя методы тестирования фасадов Laravel, мы можем написать следующий тест, чтобы проверить, что метод `Cache::get` был вызван с ожидаемым аргументом:

Pest      PHPUnit

```
use Illuminate\Support\Facades\Cache;
```

```
test('basic example', function () {
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
});
```

## Фасады против глобальных помощников

Помимо фасадов, Laravel включает в себя множество «вспомогательных» функций, которые могут выполнять общие задачи, такие как генерация шаблонов, запуск событий, запуск заданий или отправка HTTP-ответов. Многие из этих вспомогательных функций выполняют ту же функцию, что и соответствующий фасад. Например, этот вызов фасада и вызов помощника эквивалентны:

```
return Illuminate\Support\Facades\View::make('profile');

return view('profile');
```

Практической разницы между фасадами и глобальными помощниками нет абсолютно никакой. При использовании глобальных помощников вы все равно можете тестировать их точно так же, как и соответствующий фасад. Например, учитывая следующий маршрут:

```
Route::get('/cache', function () {
    return cache('key');
});
```

Помощник `cache` будет вызывать метод `get` в базовом классе, лежащем в основе фасада `Cache`. Таким образом, даже если мы используем вспомогательную функцию, мы можем написать следующий тест, чтобы убедиться, что метод был вызван с ожидаемым аргументом:

```
use Illuminate\Support\Facades\Cache;

/**
 * Отвлеченный пример функционального теста.
```

```
/*
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

## # Как фасады работают

В приложении Laravel фасад – это класс, который обеспечивает доступ к объекту из контейнера. Техника, которая выполняет эту работу, относится к классу [Facade](#). Фасады Laravel и любые пользовательские фасады, которые вы создаете, будут расширять базовый класс [Illuminate\Support\Facades\Facade](#).

Базовый класс [Facade](#) использует магический метод [\\_\\_callStatic\(\)](#), чтобы делегировать вызовы с вашего фасада объекту, извлеченному из контейнера. В приведенном ниже примере выполняется вызов кеш-системы Laravel. Взглянув на этот код, можно предположить, что статический метод [get](#) вызывается в классе [Cache](#):

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать профиль конкретного пользователя.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile(string $id): View
    {
        $user = Cache::get('user:'.$id);
```

```
        return view('profile', ['user' => $user]);
    }
}
```

Обратите внимание, что в верхней части файла мы «импортируем» фасад `Cache`. Этот фасад служит прокси для доступа к базовой реализации интерфейса `Illuminate\Contracts\Cache\Factory`. Любые вызовы, которые мы делаем с использованием фасада, будут переданы в базовый экземпляр службы кеширования Laravel.

Если мы посмотрим на этот класс `Illuminate\Support\Facades\Cache`, вы увидите, что статического метода `get` не существует:

```
class Cache extends Facade
{
    /**
     * Получить зарегистрированное имя компонента.
     */
    protected static function getFacadeAccessor(): string
    {
        return 'cache';
    }
}
```

Вместо этого фасад `Cache` расширяет базовый класс `Facade` и определяет метод `getFacadeAccessor()`. Задача этого метода – вернуть имя привязки контейнера службы. Когда пользователь ссылается на любой статический метод фасада `Cache`, Laravel извлекает объект из контейнера служб, привязанный к `cache` и запускает запрошенный метод (в данном случае `get`) этого объекта.

## # Фасады в реальном времени

Используя фасады в реальном времени, вы можете рассматривать любой класс в своем приложении, как если бы он был фасадом. Чтобы проиллюстрировать, как это можно использовать, давайте сначала рассмотрим код, который не использует фасады в реальном времени. Например, предположим, что наша модель `Podcast` имеет метод `publish`. Однако, чтобы опубликовать подкаст, нам нужно внедрить экземпляр `Publisher`:

```
<?php

namespace App\Models;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Опубликовать подкаст.
     */
    public function publish(Publisher $publisher): void
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
    }
}
```

Внедрение реализации издателя (`Publisher`) в метод позволяет нам легко тестиовать метод изолированно, поскольку мы можем имитировать внедренного издателя. Однако он требует от нас всегда передавать экземпляр издателя каждый раз, когда мы вызываем метод `publish`. Используя фасады в реальном времени, мы можем поддерживать такую же тестируемость, при этом не требуя явной передачи экземпляра `Publisher`. Чтобы сгенерировать фасад в реальном времени, добавьте к пространству имен импортируемого класса префикс `Facades`:

```
<?php

namespace App\Models;

use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Опубликовать подкаст.
     *
     * @return void
     */
    public function publish()
    {
        $this->update(['publishing' => now()]);
```

```
        Publisher::publish($this);
    }
}
```

Когда используется фасад реального времени, реализация издателя будет получена из контейнера службы с использованием той части интерфейса или имени класса, которая расположена после префикса `Facades`. При тестировании мы можем использовать встроенные в Laravel помощники для тестирования фасадов, чтобы имитировать вызов этого метода:

Pest      PHPUnit

```
<?php

use App\Models\Podcast;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('podcast can be published', function () {
    $podcast = Podcast::factory()->create();

    Publisher::shouldReceive('publish')->once()->with($podcast);

    $podcast->publish();
});
```

## # Справочник фасадов

Ниже вы найдете каждый фасад и его базовый класс. Это полезный инструмент для быстрого поиска в документации API. Ключ [привязки в контейнере служб](#) также указан, где это возможно.

Фасад	Класс	Привязка в контейнере служб
App	<a href="#">Illuminate\Foundation\Application</a>	app
Artisan	<a href="#">Illuminate\Contracts\Console\Kernel</a>	artisan

<b>Фасад</b>	<b>Класс</b>	<b>Привязка в контейнере служ</b>
Auth (Instance)	<a href="#">Illuminate\Contracts\Auth\Guard</a>	auth.driver
Auth	<a href="#">Illuminate\Auth\AuthManager</a>	auth
Blade	<a href="#">Illuminate\View\Compilers\BladeCompiler</a>	blade.compiler
Broadcast (Instance)	<a href="#">Illuminate\Contracts\Broadcasting\Broadcaster</a>	
Broadcast	<a href="#">Illuminate\Contracts\Broadcasting\Factory</a>	
Bus	<a href="#">Illuminate\Contracts\Bus\Dispatcher</a>	
Cache (Instance)	<a href="#">Illuminate\Cache\Repository</a>	cache.store
Cache	<a href="#">Illuminate\Cache\CacheManager</a>	cache
Config	<a href="#">Illuminate\Config\Repository</a>	config
Context	<a href="#">Illuminate\Log\Context\Repository</a>	
Cookie	<a href="#">Illuminate\Cookie\CookieJar</a>	cookie
Crypt	<a href="#">Illuminate\Encryption\Encrypter</a>	encrypter
Date	<a href="#">Illuminate\Support\Facades\Date</a>	date
DB (Instance)	<a href="#">Illuminate\Database\Connection</a>	db.connection
DB	<a href="#">Illuminate\Database\DatabaseManager</a>	db
Event	<a href="#">Illuminate\Events\Dispatcher</a>	events
Exceptions (Instance)	<a href="#">Illuminate\Contracts\Debug\ExceptionHandler</a>	

<b>Фасад</b>	<b>Класс</b>	<b>Привязка в контейнере служ</b>
Exceptions	<a href="#">Illuminate\Foundation\Exceptions\Handler</a>	
File	<a href="#">Illuminate\Filesystem\Filesystem</a>	files
Gate	<a href="#">Illuminate\Contracts\Auth\Access\Gate</a>	
Hash	<a href="#">Illuminate\Contracts\Hashing\Hasher</a>	hash
Http	<a href="#">Illuminate\Http\Client\Factory</a>	
Lang	<a href="#">Illuminate\Translation\Translator</a>	translator
Log	<a href="#">Illuminate\Log\LogManager</a>	log
Mail	<a href="#">Illuminate\Mail\Mailer</a>	mailer
Notification	<a href="#">Illuminate\Notifications\ChannelManager</a>	
Password (Instance)	<a href="#">Illuminate\Auth\Passwords\PasswordBroker</a>	auth.password.broker
Password	<a href="#">Illuminate\Auth\Passwords\PasswordBrokerManager</a>	auth.password
Pipeline (Instance)	<a href="#">Illuminate\Pipeline\Pipeline</a>	
Process	<a href="#">Illuminate\Process\Factory</a>	
Queue (Base Class)	<a href="#">Illuminate\Queue\Queue</a>	
Queue (Instance)	<a href="#">Illuminate\Contracts\Queue\Queue</a>	queue.connection
Queue	<a href="#">Illuminate\Queue\QueueManager</a>	queue
RateLimiter	<a href="#">Illuminate\Cache\RateLimiter</a>	

<b>Фасад</b>	<b>Класс</b>	<b>Привязка в контейнере служ</b>
Redirect	<a href="#">Illuminate\Routing\Redirector</a>	redirect
Redis (Instance)	<a href="#">Illuminate\Redis\Connections\Connection</a>	redis.connection
Redis	<a href="#">Illuminate\Redis\RedisManager</a>	redis
Request	<a href="#">Illuminate\Http\Request</a>	request
Response (Instance)	<a href="#">Illuminate\Http\Response</a>	
Response	<a href="#">Illuminate\Contracts\Routing\ResponseFactory</a>	
Route	<a href="#">Illuminate\Routing\Router</a>	router
Schedule	<a href="#">Illuminate\Console\Scheduling\Schedule</a>	
Schema	<a href="#">Illuminate\Database\Schema\Builder</a>	
Session (Instance)	<a href="#">Illuminate\Session\Store</a>	session.store
Session	<a href="#">Illuminate\Session\SessionManager</a>	session
Storage (Instance)	<a href="#">Illuminate\Contracts\Filesystem\Filesystem</a>	filesystem.disk
Storage	<a href="#">Illuminate\Filesystem\FilesystemManager</a>	filesystem
URL	<a href="#">Illuminate\Routing\UrlGenerator</a>	url
Validator (Instance)	<a href="#">Illuminate\Validation\Validator</a>	
Validator	<a href="#">Illuminate\Validation\Factory</a>	validator

<b>Фасад</b>	<b>Класс</b>	<b>Привязка в контейнере служ</b>
View (Instance)	<a href="#">Illuminate\View\View</a>	
View	<a href="#">Illuminate\View\Factory</a>	view
Vite	<a href="#">Illuminate\Foundation\Vite</a>	

# Маршрутизация

## # Основы маршрутизации

- # Файлы маршрутов по умолчанию
- # Маршруты перенаправлений
- # Маршруты представлений
- # Список ваших маршрутов
- # Настройка маршрутизации

## # Параметры маршрута

- # Обязательные параметры
- # Необязательные параметры
- # Ограничения регулярного выражения

## # Именованные маршруты

## # Группы маршрутов

- # Посредники
- # Контроллеры
- # Маршрутизация поддоменов
- # Префиксы URI сгруппированных маршрутов
- # Префиксы имен сгруппированных маршрутов

## # Привязка модели к маршруту

- # Неявная привязка
- # Неявное привязывание Enum
- # Явная привязка

## # Резервные маршруты

## # Ограничение частоты запросов

- # Определение ограничителей частоты запросов
- # Привязка ограничителей частоты запросов к маршрутам

## # Подмена методов формы

## # Доступ к текущему маршруту

## # Совместное использование ресурсов между источниками (CORS)

## # Основы маршрутизации

Простейшие маршруты Laravel принимают URI и замыкание, обеспечивая нетрудоемкий и выразительный метод определения маршрутов и поведения без сложных конфигурационных файлов маршрутизации:

```
use Illuminate\Support\Facades\Route;

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

## Файлы маршрутов по умолчанию

Все маршруты Laravel должны быть определены в файлах маршрутов, находящихся в вашем каталоге `routes`. Эти файлы автоматически загружаются Laravel с использованием конфигурации, указанной в файле `bootstrap/app.php` вашего приложения. Файл `routes/web.php` определяет маршруты для вашего веб-интерфейса. Этим маршрутам назначается [группа посредников web](#), которая обеспечивает такие функции, как состояние сессии и защита от CSRF.

Для большинства приложений вы начнете с определения маршрутов в файле `routes/web.php`. К маршрутам, определенным в `routes/web.php`, можно получить доступ, введя URL-адрес определенного маршрута в вашем браузере. Например, вы можете получить доступ к следующему маршруту, перейдя по адресу <http://example.com/user> в своем браузере:

```
use App\Http\Controllers\UserController;

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

## Маршруты API

Если ваше приложение также будет предлагать API без сохранения состояния, вы можете включить маршрутизацию API с помощью команды Artisan `install:api`:

```
php artisan install:api
```

Команда `install:api` устанавливает [Laravel Sanctum](#), который обеспечивает надежную, но простую защиту аутентификации токена API, которую можно использовать для аутентификации сторонних потребителей API, SPA, или мобильных приложений. Кроме того, команда `install:api` создает файл `routes/api.php`:

```
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');
```

Маршруты в `routes/api.php` не сохраняют состояние и назначаются [api группе посредников](#). Кроме того, к этим маршрутам автоматически применяется префикс URI `/api`, поэтому вам не нужно вручную применять его к каждому маршруту в файле. Вы можете изменить префикс, изменив файл `bootstrap/app.php` вашего приложения:

```
->withRouting(
    api: __DIR__.'/../routes/api.php',
    apiPrefix: 'api/admin',
    // ...
)
```

## Доступные методы маршрутизатора

Маршрутизатор позволяет регистрировать маршруты, отвечающие на любой HTTP-метод:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Иногда требуется зарегистрировать маршрут, отвечающий на несколько HTTP-методов. Вы можете сделать это с помощью метода `match`. Или вы даже можете

зарегистрировать маршрут, отвечающий на все HTTP-методы, используя метод `any`:

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

Route::any('/', function () {
    // ...
});
```

При определении нескольких маршрутов, которые используют один и тот же URI, маршруты, использующие методы `get`, `post`, `put`, `patch`, `delete` и `options`, должны быть определены перед маршрутами, использующими методы `any`, `match` и `redirect`. Это гарантирует, что входящий запрос соответствует правильному маршруту.

## Внедрение зависимости

Вы можете объявить любые зависимости, необходимые для вашего маршрута, в сигнатуре замыкания вашего маршрута. Объявленные зависимости будут автоматически извлечены и внедрены в замыкание с помощью [контейнера служб Laravel](#). Например, вы можете объявить класс `Illuminate\Http\Request`, чтобы текущий HTTP-запрос автоматически был внедрен в замыкание вашего маршрута:

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

## Защита от CSRF

Помните, что любые HTML-формы, указывающие на маршруты `POST`, `PUT`, `PATCH` или `DELETE`, которые определены в файле маршрутов `web`, должны включать поле токена

CSRF. В противном случае запрос будет отклонен. Вы можете прочитать больше о защите от CSRF в [документации CSRF](#):

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

## Маршруты перенаправлений

Если вы определяете маршрут, который перенаправляет на другой URI, то вы можете использовать метод `Route::redirect`. Этот метод имеет лаконичную запись, так что вам не нужно определять полный маршрут или контроллер для выполнения простого перенаправления:

```
Route::redirect('/here', '/there');
```

По умолчанию `Route::redirect` возвращает код состояния `302`. Вы можете переопределить код состояния, используя необязательный третий параметр:

```
Route::redirect('/here', '/there', 301);
```

Или вы можете использовать метод `Route::permanentRedirect` для возврата кода состояния `301`:

```
Route::permanentRedirect('/here', '/there');
```

При использовании параметров маршрута в маршрутах перенаправления, следующие параметры зарезервированы Laravel и не могут быть использованы: `destination` и `status`.

## Маршруты представлений

Если ваш маршрут должен возвращать только [HTML-шаблон](#), то вы можете использовать метод `Route::view`. Как и метод `redirect`, этот метод имеет лаконичную запись, так что вам не нужно полностью определять маршрут или контроллер. Метод `view` принимает URI в качестве первого аргумента и имя шаблона в качестве второго аргумента. Кроме того, вы можете указать массив данных для передачи в шаблон в качестве необязательного третьего аргумента:

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

При использовании параметров маршрута в маршрутах представлений, следующие параметры зарезервированы Laravel и не могут быть использованы: `view`, `data`, `status` и `headers`.

## Список ваших маршрутов

Команда Artisan `route:list` может легко предоставить обзор всех маршрутов, определенных вашим приложением:

```
php artisan route:list
```

По умолчанию промежуточное программное обеспечение маршрутов, назначенных каждому маршруту, не будет отображаться в выводе `route:list`; однако вы можете указать Laravel отображать промежуточное программное обеспечение маршрутов и имена групп промежуточного программного обеспечения, добавив опцию `-v` к команде:

```
php artisan route:list -v

# Расширить группы промежуточного программного обеспечения...
php artisan route:list -vv
```

Вы также можете указать Laravel показывать только маршруты, начинающиеся с указанного URI:

```
php artisan route:list --path=api
```

Кроме того, вы можете указать Laravel скрыть любые маршруты, определенные сторонними пакетами, предоставив опцию `--except-vendor` при выполнении команды `route:list`:

```
php artisan route:list --except-vendor
```

Также вы можете указать Laravel показывать только маршруты, определенные сторонними пакетами, предоставив опцию `--only-vendor` при выполнении команды `route:list`:

```
php artisan route:list --only-vendor
```

## Настройка маршрутизации

По умолчанию маршруты вашего приложения настраиваются и загружаются в файле `bootstrap/app.php`:

```
<?php

use Illuminate\Foundation\Application;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )->create();
```

Однако иногда вам может потребоваться определить совершенно новый файл, содержащий подмножество маршрутов вашего приложения. Для этого вы можете предоставить замыкание `then` для метода `withRouting`. В рамках этого замыкания

вы можете прописать любые дополнительные маршруты, необходимые для вашего приложения:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    web: __DIR__.'/../routes/web.php',
    commands: __DIR__.'/../routes/console.php',
    health: '/up',
    then: function () {
        Route::middleware('api')
            ->prefix('webhooks')
            ->name('webhooks.')
            ->group(base_path('routes/webhooks.php'));
    },
)
```

Или вы можете даже получить полный контроль над регистрацией маршрута, предоставив замыкание `using` для метода `withRouting`. При передаче этого аргумента платформа не будет регистрировать HTTP-маршруты, и вы несете ответственность за регистрацию всех маршрутов вручную:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    commands: __DIR__.'/../routes/console.php',
    using: function () {
        Route::middleware('api')
            ->prefix('api')
            ->group(base_path('routes/api.php'));

        Route::middleware('web')
            ->group(base_path('routes/web.php'));
    },
)
```

## # Параметры маршрута

### Обязательные параметры

Иногда бывает необходимым отслеживание сегментов URI в вашем маршруте. Например, вам может потребоваться отследить идентификатор пользователя из

URL-адреса. Вы можете сделать это, указав параметры маршрута:

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

Вы можете определить столько параметров маршрута, сколько потребуется для вашего маршрута:

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string $commentId) {
    // ...
});
```



Параметры маршрута всегда заключаются в фигурные скобки {} и должны состоять из буквенных символов. Подчеркивание (\_) также допускается в именах параметров маршрута. Параметры маршрута будут внедрены в замыкания маршрута / контроллеры в зависимости от их порядка, т.е. имена аргументов замыкания маршрута / контроллера не имеют значения.

## Параметры и внедрение зависимости

Если у вашего маршрута есть зависимости, которые вы хотите, чтобы контейнер службы Laravel автоматически внедрил в замыкание вашего маршрута, то вы должны указать эти зависимости **перед** параметрами маршрута:

```
use Illuminate\Http\Request;

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

## Необязательные параметры

Иногда может потребоваться указать параметр маршрута, который не всегда может присутствовать в URI. Вы можете сделать это, поставив знак ? после имени параметра. Не забудьте присвоить соответствующей переменной маршрута значение по умолчанию:

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

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

## Ограничения регулярного выражения

Вы можете ограничить формат параметров вашего маршрута, используя метод `where` экземпляра маршрута. Метод `where` принимает имя параметра и регулярное выражение, определяющее, как параметр должен быть ограничен:

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

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

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

Для некоторых часто используемых шаблонов регулярных выражений есть соответствующие вспомогательные методы, позволяющие быстро добавлять их к вашим маршрутам:

```
Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {
```

```
// ...
})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', CategoryEnum::cases());
```

Если входящий запрос не соответствует ограничениям шаблона маршрута, то будет возвращен **404** HTTP-ответ.

## Глобальные ограничения

Если вы хотите, чтобы параметр маршрута всегда ограничивался конкретным регулярным выражением, то вы можете использовать метод `pattern`. Вы должны определить эти шаблоны в методе `boot` класса `App\Providers\AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Route;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Route::pattern('id', '[0-9]+');
```

Как только шаблон определен, он автоматически применяется ко всем маршрутам, использующим это имя параметра:

```
Route::get('/user/{id}', function (string $id) {
    // Выполнится, только если параметр `{$id}` имеет числовое значение ...
});
```

## Кодирование обратных слешей

Компонент маршрутизации Laravel позволяет всем символам, кроме обратного слеша (/), присутствовать в значениях параметров маршрута. Вы должны явно разрешить / быть частью заполнителя {}, используя регулярное выражение условия `where`:

```
Route::get('/search/{search}', function ($search) {
    return $search;
})->where('search', '.*');
```

Обратные слеши поддерживаются только в рамках последнего сегмента маршрута.

## # Именованные маршруты

Именованные маршруты позволяют легко создавать URL-адреса или перенаправления для определенных маршрутов. Вы можете указать имя для маршрута, связав метод `name` с определением маршрута:

```
Route::get('/user/profile', function () {
    // ...
})->name('profile');
```

Вы также можете указать имена маршрутов для действий контроллера:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

Имена маршрутов всегда должны быть уникальными.

## Создание URL-адресов для именованных маршрутов

После того как вы присвоили имя указанному маршруту, вы можете использовать имя маршрута при генерации URL-адресов или перенаправлений с помощью вспомогательных глобальных функций `route` и `redirect` Laravel:

```
// Создание URL-адреса ...
$url = route('profile');

// Создание перенаправления ...
return redirect()->route('profile');

return to_route('profile');
```

Если именованный маршрут определяет параметры, то вы можете передать параметры в качестве второго аргумента функции `route`. Указанные параметры будут автоматически подставлены в сгенерированный URL в соответствующие места:

```
Route::get('/user/{id}/profile', function (string $id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1]);
```

Если вы передадите дополнительные параметры в массиве, то эти пары ключ / значение будут автоматически добавлены в строку запроса сгенерированного URL-адреса:

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

Иногда требуется указать значение по умолчанию для параметров URL запроса, например, текущий

язык. Для этого вы можете использовать метод [URL::defaults](#).

## Получение информации о текущем именованном маршруте

Если вы хотите определить, был ли текущий запрос направлен на конкретный именованный маршрут, то вы можете использовать метод [named](#) экземпляра [Route](#). Например, вы можете проверить имя текущего маршрута из посредника маршрута:

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Обработка входящего запроса.
 * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

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

## # Группы маршрутов

Группы маршрутов позволяют совместно использовать атрибуты маршрута (например, посредники) для большого количества маршрутов без необходимости определять эти атрибуты для каждого маршрута отдельно.

Вложенные группы пытаются разумно «объединить» атрибуты со своей родительской группой. Посредники и условия [where](#) объединяются, а имена и префиксы добавляются. Разделители пространства имен и слеши в префиксах URI автоматически добавляются там, где это необходимо.

## Посредники

Чтобы назначить [посредника](#) всем маршрутам в группе, вы можете использовать метод `middleware` перед определением группы. Посредники будут выполняться в том порядке, в котором они перечислены в массиве:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Использует посредники `first` и `second` ...
    });
    Route::get('/user/profile', function () {
        // Использует посредники `first` и `second` ...
    });
});
```

## Контроллеры

Если группа маршрутов использует один и тот же [контроллер](#), вы можете использовать метод `controller` для определения общего контроллера всех маршрутов в группе. Затем при определении маршрутов вам нужно будет указать только метод вызываемого контроллера:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

## Маршрутизация поддоменов

Группы маршрутов также могут использоваться для управления маршрутизацией поддоменов. Поддоменам могут быть назначены параметры маршрута так же, как и URI маршрута, что позволяет вам отследить сегмент с поддоменом для использования его в вашем маршруте или контроллере. Поддомен можно указать, вызвав метод `domain` перед определением группы:

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('/user/{id}', function (string $account, string $id) {
        // ...
    });
});
```

```
});  
});
```

Чтобы обеспечить доступность маршрутов поддоменов, вы должны зарегистрировать маршруты поддоменов перед регистрацией маршрутов корневого домена. Это предотвратит перезапись маршрутами корневого домена маршрутов поддоменов, имеющих одинаковый путь URI.

## Префиксы URI сгруппированных маршрутов

Метод `prefix` используется для подстановки указанного URI в качестве префикса каждому маршруту в группе. Например, можно подставить префикс `admin` перед всеми URI сгруппированных маршрутов:

```
Route::prefix('admin')->group(function () {  
    Route::get('/users', function () {  
        // Соответствует URL-адресу `/admin/users` ...  
    });  
});
```

## Префиксы имен сгруппированных маршрутов

Метод `name` может быть использован для добавления префикса к каждому имени маршрута в группе с использованием заданной строки. Например, вы можете захотеть добавить префикс `admin` ко всем именам маршрутов в группе. Заданная строка добавляется к имени маршрута точно так, как она указана, поэтому мы обязательно предоставим знак `.` в конце префикса:

```
Route::name('admin.')->group(function () {  
    Route::get('/users', function () {  
        // Маршруту присвоено имя `admin.users` ...  
    })->name('users');  
});
```

# # Привязка модели к маршруту

При внедрении идентификатора модели в маршрут или действие контроллера вы часто будете запрашивать базу данных, чтобы получить модель с соответствующим идентификатором. Привязка модели к маршруту Laravel обеспечивает удобный способ автоматического внедрения экземпляров модели непосредственно в ваши маршруты. Например, вместо того, чтобы внедрять идентификатор пользователя, вы можете внедрить весь экземпляр модели `User` с соответствующим идентификатором.

## Неявная привязка

Laravel автоматически извлечет модели Eloquent, определенные в маршрутах или действиях контроллера, чьи имена переменных объявленного типа соответствуют имени сегмента маршрута. Например:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Так как переменная `$user` типизирована как модель `App\Models\User` Eloquent и имя переменной соответствует сегменту `{user}` URI, то Laravel автоматически внедрит экземпляр модели с идентификатором, совпадающим со значением URI из запроса. Если соответствующий экземпляр модели не найден в базе данных, то автоматически будет сгенерирован `404` HTTP-ответ.

Конечно, неявная привязка также возможна при использовании методов контроллера. Опять же, обратите внимание, что сегмент `{user}` URI соответствует переменной `$user` в контроллере, которая типизирована как `App\Models\User`:

```
use App\Http\Controllers\UserController;
use App\Models\User;

// Определение маршрута ...
Route::get('/users/{user}', [UserController::class, 'show']);

// Определение метода контроллера ...
public function show(User $user)
{
```

```
        return view('user.profile', ['user' => $user]);
    }
}
```

## Программное удаление моделей

Как правило, неявная привязка модели не будет извлекать модели, которые были [удалены программно](#). Однако вы можете указать неявной привязке извлекать эти модели, привязав метод `withTrashed` к определению вашего маршрута:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

## Изменение ключа по умолчанию

По желанию можно извлекать модели Eloquent, используя столбец, отличный от `id`. Для этого вы можете указать столбец в определении параметра маршрута:

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

Если вы хотите, чтобы при извлечении класса связанной модели всегда использовался столбец базы данных, отличный от `id`, то вы можете переопределить метод `getRouteKeyName` модели Eloquent:

```
/**
 * Получить ключ маршрута для модели.
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

## Измененный ключ и ограничения неявной привязки модели

При неявном связывании нескольких моделей Eloquent в одном определении маршрута бывает необходимо ограничить вторую модель Eloquent так, чтобы она была дочерней по отношению к предыдущей модели Eloquent. Например, рассмотрим это определение маршрута, которое извлекает пост в блоге по `slug` для конкретного пользователя:

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
});
```

При использовании неявной привязки с измененным ключом в качестве параметра вложенного маршрута, Laravel автоматически задает ограничение запроса для получения вложенной модели своим родителем, используя соглашения, чтобы угадать имя отношения родительской модели. В этом случае предполагается, что модель `User` имеет отношение с именем `posts` (форма множественного числа имени параметра маршрута), которое можно использовать для получения модели `Post`.

При желании вы можете указать Laravel охватывать “дочерние” привязки, даже если пользовательский ключ не предоставлен. Для этого вы можете вызвать метод `scopeBindings` при определении своего маршрута:

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
    return $post;
})->scopeBindings();
```

Или вы можете указать целой группе определений маршрутов использовать привязки с заданной областью действия:

```
Route::scopeBindings()->group(function () {
    Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
        return $post;
    });
});
```

Точно так же, вы можете явно указать Laravel не использовать ограничение области действия привязок, вызвав метод `withoutScopedBindings`:

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
})->withoutScopedBindings();
```

## Настройка поведения при отсутствии модели

Обычно, если неявно связанная модель не найдена, то генерируется `404` HTTP-ответ. Однако вы можете изменить это поведение, вызвав метод `missing` при определении вашего маршрута. Метод `missing` принимает замыкание, которое будет вызываться, если неявно связанная модель не может быть найдена:

```
use App\Http\Controllers\LocationsController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class, 'show'])
    ->name('locations.view')
    ->missing(function (Request $request) {
        return Redirect::route('locations.index');
});
```

## Неявное привязывание Enum

PHP 8.1 ввёл поддержку [Enum \(Перечислений\)](#). Чтобы дополнить эту функцию, Laravel позволяет указывать [поддерживаемый Enum](#) в определении маршрута, и Laravel будет вызывать маршрут только в том случае, если сегмент маршрута соответствует допустимому значению Enum. В противном случае автоматически будет возвращен ответ HTTP 404. Например, учитывая следующий Enum:

```
<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}
```

Вы можете определить маршрут, который будет вызываться только в случае, если сегмент `{category}` маршрута является `fruits` или `people`. В противном случае Laravel вернет ответ HTTP 404:

```
use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});
```

## Явная привязка

Вам необязательно использовать неявные привязки модели на основе соглашений Laravel, чтобы использовать привязку модели. Вы также можете явно определить, как параметры маршрута должны быть сопоставлены моделям. Чтобы зарегистрировать явную привязку, используйте метод маршрутизатора `model`, чтобы указать класс для переданного параметра. Вы должны определить ваши явные привязки модели в начале метода `boot` вашего класса `AppServiceProvider`:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Route::model('user', User::class);
}
```

Затем определите маршрут, содержащий параметр `{user}`:

```
use App\Models\User;

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

Поскольку мы связали все параметры `{user}` с моделью `App\Models\User`, экземпляр этого класса будет внедрен в маршрут. Так, например, при запросе `users/1` будет внедрен экземпляр `User` из базы данных с идентификатором `1`.

Если соответствующий экземпляр модели не найден в базе данных, то автоматически будет сгенерирован `404` HTTP-ответ.

## Изменение логики связывания

Если вы хотите определить свою собственную логику связывания модели, то вы можете использовать метод `Route::bind`. Замыкание, которое вы передаете методу `bind`, получит значение сегмента URI и должно вернуть экземпляр класса, который должен быть внедрен в маршрут. Опять же, эти изменения должны выполняться в методе `boot` поставщика `AppServiceProvider` вашего приложения:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Route::bind('user', function (string $value) {
        return User::where('name', $value)->firstOrFail();
    });
}
```

В качестве альтернативы вы можете переопределить метод `resolveRouteBinding` вашей модели Eloquent. Этот метод получит значение сегмента URI и должен вернуть экземпляр класса, который должен быть внедрен в маршрут:

```
/**
 * Получить модель для привязанного к маршруту значения параметра.
 *
 * @param mixed $value
 * @param string|null $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value, $field = null)
{
```

```
        return $this->where('name', $value)->firstOrFail();
    }
}
```

Если в маршруте используется [ограничения неявной привязки модели](#), то для получения связанной дочерней модели будет использоваться метод `resolveChildRouteBinding` родительской модели:

```
/**
 * Получить дочернюю модель для привязанного к маршруту значения параметра.
 *
 * @param string $childType
 * @param mixed $value
 * @param string|null $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveChildRouteBinding($childType, $value, $field)
{
    return parent::resolveChildRouteBinding($childType, $value, $field);
}
```

## # Резервные маршруты

Используя метод `Route::fallback`, вы можете определить маршрут, который будет выполняться, когда ни один другой маршрут не соответствует входящему запросу. Как правило, необработанные запросы автоматически отображают страницу [404](#) через обработчик исключений вашего приложения. Однако, поскольку вы обычно определяете резервный маршрут в своем файле `routes/web.php`, то все посредники группы `web` будут применены к этому маршруту. При необходимости вы можете добавить дополнительный посредник для этого маршрута:

```
Route::fallback(function () {
    // ...
});
```

Резервный маршрут всегда должен быть последним зарегистрированным маршрутом в вашем приложении.

# # Ограничение частоты запросов

## Определение ограничителей частоты запросов

Laravel включает в себя мощные и настраиваемые сервисы для ограничения частоты запросов, которые вы можете использовать для ограничения объема трафика для определенного маршрута или группы маршрутов. Чтобы начать, вам следует определить конфигурации ограничителей частоты запросов, соответствующие потребностям вашего приложения.

Ограничители скорости могут быть определены в методе `boot` класса `App\Providers\AppServiceProvider` вашего приложения:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Запуск любых служб приложения.
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
    });
}
```

Ограничители определяются с помощью метода `for` фасада `RateLimiter`. Метод `for` принимает имя ограничителя и замыкание, которое возвращает конфигурацию ограничений, применяемых к назначенным маршрутам. Конфигурация ограничений – это экземпляры класса `Illuminate\Cache\RateLimiting\Limit`. Этот класс содержит полезные методы «построения», чтобы вы могли быстро определить свой «лимит». Имя ограничителя может быть любой строкой по вашему желанию:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Запуск любых служб приложения.
 */
protected function boot(): void
```

```
{  
    RateLimiter::for('global', function (Request $request) {  
        return Limit::perMinute(1000);  
    });  
}
```

Если входящий запрос превышает указанный лимит, то Laravel автоматически вернет ответ с `429` кодом состояния HTTP. Если вы хотите определить свой собственный ответ, который должен возвращаться, то вы можете использовать метод `response`:

```
RateLimiter::for('global', function (Request $request) {  
    return Limit::perMinute(1000)->response(function (Request $request, array $headers) {  
        return response('Custom response...', 429, $headers);  
    });  
});
```



Поскольку замыкание получает экземпляр входящего HTTP-запроса, то вы можете динамически создать ограничение на основе входящего запроса или статуса аутентификации пользователя:

```
RateLimiter::for('uploads', function (Request $request) {  
    return $request->user()->vipCustomer()  
        ? Limit::none()  
        : Limit::perMinute(100);  
});
```

## Сегментация ограничений частоты запросов

Иногда может потребоваться сегментация ограничений по некоторым произвольным значениям. Например, вы можете разрешить пользователям получать доступ к указанному маршруту 100 раз в минуту на каждый IP-адрес. Для этого можно использовать метод `by` при построении лимита:

```
RateLimiter::for('uploads', function (Request $request) {  
    return $request->user()->vipCustomer()  
        ? Limit::none()  
        : Limit::perMinute(100)->by($request->ip());  
});
```

Чтобы проиллюстрировать эту функцию на другом примере, мы можем ограничить доступ к маршруту до 100 раз в минуту для каждого аутентифицированного ID пользователя или 10 раз в минуту для каждого IP-адреса для гостей:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()
        ? Limit::perMinute(100)->by($request->user()->id)
        : Limit::perMinute(10)->by($request->ip());
});
```

## Множественные ограничения частоты запросов

При необходимости вы можете вернуть массив ограничений при указании конфигурации ограничителя. Каждое ограничение будет оцениваться для маршрута в зависимости от порядка, в котором они размещены в массиве:

```
RateLimiter::for('login', function (Request $request) {
    return [
        Limit::perMinute(500),
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

Если вы назначаете несколько ограничений скорости, сегментированных по одинаковым значениям `by`, вам следует убедиться, что каждое значение `by` уникально. Самый простой способ добиться этого — добавить префикс к значениям, заданным методом `by`:

```
RateLimiter::for('uploads', function (Request $request) {
    return [
        Limit::perMinute(10)->by('minute:'.$request->user()->id),
        Limit::perDay(1000)->by('day:'.$request->user()->id),
    ];
});
```

## Привязка ограничителей частоты запросов к маршрутам

Ограничители могут быть закреплены за маршрутами или группами маршрутов с помощью [посредника `throttle`](#). Посредник `throttle` принимает имя ограничителя,

которое вы хотите назначить маршруту:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        // ...
    });

    Route::post('/video', function () {
        // ...
    });
});
```

## Использование Redis для посредника throttle

По умолчанию посредник `throttle` сопоставлен классу

`Illuminate\Routing\Middleware\ThrottleRequests`. Однако, если вы используете Redis в качестве драйвера кэша вашего приложения, вы можете поручить Laravel использовать Redis для управления ограничением скорости. Для этого вам следует использовать метод `throttleWithRedis` в файле `bootstrap/app.php` вашего приложения. Этот метод сопоставляет посредника `throttle` с классом посредника

`Illuminate\Routing\Middleware\ThrottleRequestsWithRedis`:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->throttleWithRedis();
    // ...
})
```

## # Подмена методов формы

HTML-формы не поддерживают действия `PUT`, `PATCH` или `DELETE`. Таким образом, при определении маршрутов `PUT`, `PATCH` или `DELETE`, которые вызываются из HTML-формы, вам нужно будет добавить в форму скрытое поле `_method`. Значение, отправленное с полем `_method`, будет использоваться как метод HTTP-запроса:

```
<form action="/example" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

Для удобства вы можете использовать директиву `@method` [шаблонизатора Blade](#) для создания поля ввода `_method`:

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

## # Доступ к текущему маршруту

Вы можете использовать методы `current`, `currentRouteName` и `currentRouteAction` фасада `Route` для доступа к информации о маршруте, обрабатывающем входящий запрос:

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

Вы можете обратиться к документации API [базового класса фасада Route](#) и [экземпляра Route](#), чтобы просмотреть все методы, доступные в классах маршрутизатора и маршрута.

## # Совместное использование ресурсов между источниками (CORS)

Laravel может автоматически отвечать на HTTP-запросы CORS `OPTIONS` значениями, которые вы сконфигурируете. Запросы `OPTIONS` будут автоматически обрабатываться [HandleCors посредником](#), которое автоматически включается в глобальный стек промежуточного программного обеспечения вашего приложения.

Иногда вам может потребоваться настроить значения конфигурации CORS для вашего приложения. Вы можете сделать это, опубликовав файл конфигурации `cors` с помощью Artisan-команды `config:publish`:

```
php artisan config:publish cors
```

Эта команда поместит файл конфигурации `cors.php` в каталог `config` вашего приложения.

Для получения дополнительной информации о CORS и заголовках CORS обратитесь к [веб-документации MDN по CORS](#).

## # Кеширование маршрутов

При развертывании вашего приложения на рабочем веб-сервере, вы должны воспользоваться кешем маршрутов Laravel. Использование кеша маршрутов резко сократит время, необходимое для регистрации всех маршрутов вашего приложения. Чтобы сгенерировать кеш маршрута, выполните команду `route:cache` Artisan:

```
php artisan route:cache
```

После выполнения этой команды ваш файл кеша маршрутов будет загружаться при каждом запросе. Помните, что если вы добавляете какие-либо новые маршруты, то вам нужно будет сгенерировать новый кеш маршрутов. По этой причине вы должны запускать команду `route:cache` только во время развертывания вашего проекта.

Вы можете использовать команду `route:clear` для очистки кеша маршрута:

```
php artisan route:clear
```

# Посредники (middleware)

- # Введение
- # Определение посредника
- # Регистрация посредника
  - # Глобальный стек HTTP-посредников
  - # Назначение посредников маршрутам
  - # Группы посредников
  - # Псевдонимы посредников
  - # Сортировка посредников
- # Параметры посредника
- # Завершающий посредник

## # Введение

Посредник обеспечивает удобный механизм для проверки и фильтрации HTTP-запросов, поступающих в ваше приложение. Например, в Laravel уже содержится посредник, проверяющий аутентификацию пользователя вашего приложения. Если пользователь не аутентифицирован, то посредник перенаправит пользователя на экран входа в ваше приложение. Однако, если пользователь аутентифицирован, то посредник позволит запросу продолжить работу в приложении.

Посредник может быть написан для выполнения различных задач помимо аутентификации. Например, посредник для ведения журналов может регистрировать все входящие запросы к вашему приложению. В Laravel включено множество посредников, включая посредники для аутентификации и защиты CSRF; однако все определяемые пользователями посредники обычно находятся в каталоге `app/Http/Middleware` вашего приложения.

## # Определение посредника

Чтобы создать нового посредника, используйте команду `make:middleware Artisan`:

```
php artisan make:middleware EnsureTokenIsValid
```

Эта команда поместит новый класс посредника в каталог `app/Http/Middleware` вашего приложения. В этом посреднике мы будем разрешать доступ к маршруту только в том случае, если значение входящего `token` соответствует указанному. В противном случае мы перенаправим пользователя по маршруту `/home`:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Обработка входящего запроса.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('/home');
        }

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

Как видите, если переданный `token` не совпадает с нашим секретным токеном, то посредник вернет клиенту HTTP-перенаправление; в противном случае запрос будет передан в приложение. Чтобы передать запрос дальше в приложение (позволяя «пройти» посредника), вы должны вызвать замыкание `$next` с параметром `$request`.

Лучше всего представить себе посредников как серию «слоев» для HTTP-запроса, которые необходимо пройти, прежде чем запрос попадет в ваше приложение. Каждый слой может рассмотреть запрос и даже полностью отклонить его.

Все посредники извлекаются из [контейнера служб](#), поэтому вы можете объявить необходимые вам зависимости в конструкторе посредника.

## Посредники и ответы

Конечно, посредник может выполнять задачи до или после передачи запроса в приложение. Например, следующий посредник будет выполнять некоторую задачу **до** того, как запрос будет обработан приложением:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // Выполнить действие

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

Однако, этот посредник будет выполнять свою задачу **после** обработки входящего запроса приложением:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
```

```
public function handle(Request $request, Closure $next): Response
{
    $response = $next($request);

    // Выполнить действие

    return $response;
}
}
```

## # Регистрация посредника

### Глобальный стек HTTP-посредников

Если вы хотите, чтобы посредник запускался при каждом HTTP-запросе к вашему приложению, вы можете добавить его в глобальный стек посредников в файле `bootstrap/app.php` вашего приложения:

```
use App\Http\Middleware\EnsureTokenIsValid;

->withMiddleware(function (Middleware $middleware) {
    $middleware->append(EnsureTokenIsValid::class);
})
```

Объект `$middleware`, предоставляемый замыканию `withMiddleware`, является экземпляром `Illuminate\Foundation\Configuration\Middleware` и отвечает за управление посредником, назначенным маршрутам вашего приложения. Метод `append` добавляет посредника в конец глобального списка посредников. Если вы хотите добавить посредника в начало списка, вам следует использовать метод `prepend`.

### Ручное управление глобальным стеком посредников

Если вы хотите управлять глобальным стеком посредников Laravel вручную, вы можете предоставить глобальный стек посредников Laravel по умолчанию для метода `use`. Затем вы можете при необходимости настроить стек посредников по умолчанию:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->use([

```

```
\Illuminate\Foundation\Http\Middleware\InvokeDeferredCallbacks::class,
// \Illuminate\Http\Middleware\TrustHosts::class,
\Illuminate\Http\Middleware\TrustProxies::class,
\Illuminate\Http\Middleware\HandleCors::class,
\Illuminate\Foundation\Http\Middleware\PreventRequestsDuringMaintenance::class,
\Illuminate\Http\Middleware\ValidatePostSize::class,
\Illuminate\Foundation\Http\Middleware\TrimStrings::class,
\Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
]);
})
```

## Назначение посредников маршрутам

Если вы хотите назначить посредника (middleware) для определенных маршрутов, вы можете использовать метод `middleware` при определении маршрута:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    // ...
})->middleware(EnsureTokenIsValid::class);
```

Вы также можете назначить несколько middleware для маршрута, передав массив имен в метод `middleware`:

```
Route::get('/', function () {
    // ...
})->middleware([First::class, Second::class]);
```

## Исключение посредников

При назначении посредника группе маршрутов, иногда может потребоваться запретить применение посредника к одному из маршрутов в группе. Вы можете сделать это с помощью метода `withoutMiddleware`:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::middleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/', function () {
        // ...
    });
});
```

```
Route::get('/profile', function () {
    // ...
})->withoutMiddleware([EnsureTokenIsValid::class]);
});
```

Вы также можете исключить данный набор посредников из всей [группы маршрутов](#):

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

Метод `withoutMiddleware` удаляет только посредника маршрутизации и не применим к [глобальному посреднику](#).

## Группы посредников

По желанию можно сгруппировать несколько посредников под одним ключом, чтобы упростить их назначение маршрутам. Вы можете сделать это, используя метод `appendToGroup` в файле `bootstrap/app.php` вашего приложения:

```
use App\Http\Middleware\First;
use App\Http\Middleware\Second;

->withMiddleware(function (Middleware $middleware) {
    $middleware->appendToGroup('group-name', [
        First::class,
        Second::class,
    ]);

    $middleware->prependToGroup('group-name', [
        First::class,
        Second::class,
    ]);
})
```

Группы посредников могут быть назначены маршрутам и действиям контроллера, используя тот же синтаксис, что и с индивидуальным посредником:

```
Route::get('/', function () {
    // ...
})->middleware('group-name');

Route::middleware(['group-name'])->group(function () {
    // ...
});
```

## Группы посредников Laravel по умолчанию

Laravel включает в себя предопределенные группы посредников `web` и `api`, которые содержат общие посредники, которое вы, возможно, захотите применить к своим веб и API маршрутам. Помните, что Laravel автоматически применяет эти группы посредников к соответствующим файлам `routes/web.php` и `routes/api.php`:

### Группа посредников `web`

```
Illuminate\Cookie\Middleware\EncryptCookies  
Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse  
Illuminate\Session\Middleware\StartSession  
Illuminate\View\Middleware\ShareErrorsFromSession  
Illuminate\Foundation\Http\Middleware\ValidateCsrfToken  
Illuminate\Routing\Middleware\SubstituteBindings
```

### Группа посредников `api`

```
Illuminate\Routing\Middleware\SubstituteBindings
```

Если вы хотите добавить или добавить посредника к этим группам, вы можете использовать методы `web` и `api` в файле `bootstrap/app.php` вашего приложения.

Методы `web` и `api` являются удобной альтернативой методу `appendToGroup`:

```
use App\Http\Middleware\EnsureTokenIsValid;
use App\Http\Middleware\EnsureUserIsSubscribed;

->withMiddleware(function (Middleware $middleware) {
    $middleware->web(append: [
        EnsureUserIsSubscribed::class,
    ]);

    $middleware->api(prepend: [
        EnsureTokenIsValid::class,
    ]);
})
```

Вы даже можете заменить одну из записей группы посредников Laravel по умолчанию на собственного посредника:

```
use App\Http\Middleware\StartCustomSession;
use Illuminate\Session\Middleware\StartSession;

$middleware->web(replace: [
    StartSession::class => StartCustomSession::class,
]);
```

Или вы можете полностью удалить посредника:

```
$middleware->web(remove: [
    StartSession::class,
]);
```

## Ручное управление группами посредников Laravel по умолчанию

Если вы хотите вручную управлять всеми посредниками в группах посредников Laravel по умолчанию `web` и `api`, вы можете полностью переопределить эти группы. В приведенном ниже примере будут определены группы посредников `web` и `api` с их посредниками по умолчанию, что позволит вам настроить их по мере необходимости:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->group('web', [
```

```
\Illuminate\Cookie\Middleware\EncryptCookies::class,
\Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
\Illuminate\Session\Middleware\StartSession::class,
\Illuminate\View\Middleware\ShareErrorsFromSession::class,
\Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,
\Illuminate\Routing\Middleware\SubstituteBindings::class,
// \Illuminate\Session\Middleware\AuthenticateSession::class,
]);
}

$middleware->group('api', [
    // \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    // 'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
]);
})
```

По умолчанию группы посредников `web` и `api` автоматически применяются к соответствующим файлам `routes/web.php` и `routes/api.php` вашего приложения с помощью файла `bootstrap/app.php`.

## Псевдонимы посредников

Вы можете назначить псевдонимы посредникам в файле `bootstrap/app.php` вашего приложения. Псевдонимы посредников позволяют определить короткий псевдоним для данного класса посредника, что может быть особенно полезно для посредника с длинными именами классов:

```
use App\Http\Middleware\EnsureUserIsSubscribed;

->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'subscribed' => EnsureUserIsSubscribed::class
    ]);
})
```

После того как псевдоним посредника определен в файле `bootstrap/app.php` вашего приложения, вы можете использовать его при назначении посредника маршрутам:

```
Route::get('/profile', function () {
    // ...
})->middleware('subscribed');
```

Для удобства некоторые встроенные посредники Laravel по умолчанию имеют псевдонимы. Например, посредник `auth` является псевдонимом посредника `Illuminate\Auth\Middleware\Authenticate`. Ниже приведен список псевдонимов посредников по умолчанию:

Псевдоним	Посредник
<code>auth</code>	<code>Illuminate\Auth\Middleware\Authenticate</code>
<code>auth.basic</code>	<code>Illuminate\Auth\Middleware\AuthenticateWithBasicAuth</code>
<code>auth.session</code>	<code>Illuminate\Session\Middleware\AuthenticateSession</code>
<code>cache.headers</code>	<code>Illuminate\Http\Middleware\SetCacheHeaders</code>
<code>can</code>	<code>Illuminate\Auth\Middleware\Authorize</code>
<code>guest</code>	<code>Illuminate\Auth\Middleware\RedirectIfAuthenticated</code>
<code>password.confirm</code>	<code>Illuminate\Auth\Middleware\RequirePassword</code>
<code>precognitive</code>	<code>Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests</code>
<code>signed</code>	<code>Illuminate\Routing\Middleware\ValidateSignature</code>
<code>subscribed</code>	<code>\Spark\Http\Middleware\VerifyBillableIsSubscribed</code>
<code>throttle</code>	<code>Illuminate\Routing\Middleware\ThrottleRequests</code> ИЛИ <code>Illuminate\Routing\Middleware\ThrottleRequestsWithRedis</code>
<code>verified</code>	<code>Illuminate\Auth\Middleware\EnsureEmailIsVerified</code>

## Сортировка посредников

В редких случаях вам может потребоваться, чтобы ваши посредники выполнялись в определенном порядке, но вы не можете контролировать их порядок, когда они

назначены маршруту. В этом случае вы можете указать приоритет посредников, используя метод `priority` в файле `bootstrap/app.php` вашего приложения:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->priority([
        \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,
        \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
        \Illuminate\Routing\Middleware\ThrottleRequests::class,
        \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
        \Illuminate\Auth\Middleware\Authorize::class,
    ]);
})
```

## # Параметры посредника

Посредник также может получать дополнительные параметры. Например, если вашему приложению необходимо проверить, что аутентифицированный пользователь имеет конкретную «роль» перед выполнением им конкретного действия, то вы можете создать посредника, например, `EnsureUserHasRole`, который получит имя роли в качестве дополнительного аргумента.

Дополнительные параметры посредника будут переданы после аргумента `$next`:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureUserHasRole
{
    /**
     * Обработка входящего запроса.
}
```

```
* @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
*/
public function handle(Request $request, Closure $next, string $role): Response
{
    if (! $request->user()->hasRole($role)) {
        // Перенаправление ...
    }

    return $next($request);
}

}
```

Параметры посредника можно указать при определении маршрута, разделив имя посредника и параметры символом `:`.

```
use App\Http\Middleware\EnsureUserHasRole;

Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware(EnsureUserHasRole::class.':editor');
```

Несколько параметров следует разделять запятыми:

```
Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware(EnsureUserHasRole::class.':editor,publisher');
```

## # Завершающий посредник

Иногда посреднику может потребоваться выполнить некоторую работу после отправки HTTP-ответа в браузер. Если вы определите метод `terminate` в своем посреднике и при условии, что ваш веб-сервер использует FastCGI, то метод `terminate` будет автоматически вызван после отправки ответа в браузер:

```
<?php
namespace Illuminate\Session\Middleware;
```

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class TerminatingMiddleware
{
    /**
     * Обработка входящего запроса.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);
    }

    /**
     * Обработать задачи после отправки ответа в браузер.
     */
    public function terminate(Request $request, Response $response): void
    {
        // ...
    }
}
```

Метод `terminate` должен получать и запрос, и ответ. После того как вы определили завершающий посредник, вы должны добавить его в список маршрутов или глобальный стек посредников в файле `bootstrap/app.php` вашего приложения.

При вызове метода `terminate` посредника, Laravel извлечет новый экземпляр посредника из [контейнера служб](#). Если вы хотите использовать один и тот же экземпляр посредника при вызове методов `handle` и `terminate`, то зарегистрируйте посредника в контейнере, используя метод контейнера `singleton`. Обычно это должно быть сделано в методе `register` вашего `AppServiceProvider`:

```
use App\Http\Middleware\TerminatingMiddleware;

/**
 * Регистрация любых служб приложения.
 */
public function register(): void
{
    $this->app->singleton(TerminatingMiddleware::class);
}
```

# Предотвращение атак CSRF

# Введение

# Предотвращение запросов CSRF

# CSRF-токены и SPA-приложения

# Исключение URI из защиты от CSRF

# Токен X-CSRF

# Токен X-XSRF

## # Введение

Межсайтовая подделка запроса – это разновидность вредоносного эксплойта, при котором неавторизованные команды выполняются от имени аутентифицированного пользователя. К счастью, Laravel позволяет легко защитить ваше приложение от [Межсайтовой подделки запроса](#) ([Cross Site Request Forgery](#) – CSRF).

## Объяснение уязвимости

Если вы не знакомы с Межсайтовой подделкой запросов, то давайте обсудим пример того, как можно использовать эту уязвимость. Представьте, что ваше приложение имеет маршрут `/user/email`, который принимает `POST`-запрос для изменения адреса электронной почты аутентифицированного пользователя. Скорее всего, этот маршрут ожидает, что поле ввода `email` будет содержать адрес электронной почты, который пользователь хотел бы начать использовать.

Без защиты от CSRF вредоносный веб-сайт может создать HTML-форму, которая указывает на маршрут вашего приложения `/user/email` и отправляет собственный адрес электронной почты злоумышленника:

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

<script>
```

```
document.forms[0].submit();
</script>
```

Если вредоносный веб-сайт автоматически отправляет форму при загрузке страницы, злоумышленнику нужно только подтолкнуть ничего не подозревающего пользователя вашего приложения посетить свой веб-сайт, и его адрес электронной почты будет изменен в вашем приложении.

Чтобы предотвратить эту уязвимость, нам необходимо проверять каждый входящий запрос `POST`, `PUT`, `PATCH`, или `DELETE` на секретное значение сессии, к которому вредоносное приложение не может получить доступ.

## # Предотвращение запросов CSRF

Laravel автоматически генерирует «токен» CSRF для каждой активной [пользовательской сессии](#), управляемой приложением. Этот токен используется для проверки того, что аутентифицированный пользователь действительно является лицом, выполняющим запросы к приложению. Поскольку этот токен хранится в сессии пользователя и изменяется каждый раз при повторном создании сессии, вредоносное приложение не может получить к нему доступ.

К CSRF-токену текущей сессии можно получить доступ через сессию запроса или с помощью глобального помощника `csrf_token`:

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

Каждый раз, когда вы создаете HTML-форму «POST», «PUT», «PATCH» или «DELETE» в своем приложении, вы должны включать в форму скрытое поле `_token` CSRF, чтобы посредник CSRF мог проверить запрос. Для удобства вы можете использовать директиву Blade `@csrf` для создания скрытого поля ввода, содержащего токен:

```
<form method="POST" action="/profile">
    @csrf

    <!-- Эквивалентно ... -->
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
</form>
```

Посредник [Illuminate\Foundation\Http\Middleware\ValidateCsrfToken](#), который по умолчанию стоит в группе посредников [web](#), автоматически проверяет соответствие токена во входном запросе и токен, хранящийся в сессии. Когда эти два токена совпадают, мы знаем, что запрос инициирует аутентифицированный пользователь.

## CSRF-токены и SPA-приложения

Если вы создаете SPA, который использует Laravel в качестве серверной части API, вам следует обратиться к [документации Laravel Sanctum](#) для получения информации об аутентификации с помощью вашего API и защите от уязвимостей CSRF.

## Исключение URI из защиты от CSRF

По желанию можно исключить набор URI из защиты от CSRF. Например, если вы используете [Stripe](#) для обработки платежей и используете их систему веб-хуков, вам нужно будет исключить маршрут обработчика веб-хуков Stripe из защиты от CSRF, поскольку Stripe не будет знать, какой токен CSRF отправить вашим маршрутам.

Как правило, вы должны размещать эти виды маршрутов вне группы посредников [web](#), которую Laravel применяет ко всем маршрутам в файле [routes/web.php](#). Однако вы также можете исключить определенные маршруты, указав их URI методу [validateCsrfTokens](#) в файле [bootstrap/app.php](#) вашего приложения:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->validateCsrfTokens(except: [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ]);
})
```

Для удобства посредник CSRF автоматически отключается для всех маршрутов при [выполнении тестов](#).

## # Токен X-CSRF

В дополнение к проверке токена CSRF в качестве параметра POST-запроса посредник `Illuminate\Foundation\Http\Middleware\ValidateCsrfToken`, который по умолчанию включен в группу посредников `web`, также проверяет заголовок запроса `X-CSRF-TOKEN`. Вы можете, например, сохранить токен в HTML-теге `meta`:

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

Затем, вы можете указать библиотеке, такой как jQuery, автоматически добавлять токен во все заголовки запросов. Это обеспечивает простую и удобную защиту от CSRF для ваших приложений с использованием устаревшей технологии JavaScript на основе AJAX:

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

## # Токен X-XSRF

Laravel хранит текущий токен CSRF в зашифрованном файле Cookies `XSRF-TOKEN`, который содержится в каждом ответе, генерируемым фреймворком. Вы можете использовать значение Cookies для установки заголовка запроса `X-XSRF-TOKEN`.

Этот файл Cookies, в первую очередь, отправляется для удобства разработчика, поскольку некоторые фреймворки и библиотеки JavaScript, такие, как Angular и Axios, автоматически помещают его значение в заголовок `X-XSRF-TOKEN` в запросах с одним и тем же источником.

По умолчанию файл `resources/js/bootstrap.js`  
включает HTTP-библиотеку Axios, которая  
автоматически отправляет заголовок `X-XSRF-TOKEN`.

# Контроллеры

## # Введение

## # Написание контроллеров

# Основные понятия о контроллерах

# Контроллеры одиночного действия

## # Посредник контроллера

## # Ресурсные контроллеры

# Частичные ресурсные маршруты

# Вложенные ресурсы

# Именование ресурсных маршрутов

# Именование параметров ресурсных маршрутов

# Ограничение ресурсных маршрутов

# Локализация URI ресурсов

# Дополнение ресурсных контроллеров

# Синглтон-ресурс контроллеров

## # Внедрение зависимостей и контроллеры

## # Введение

Вместо того чтобы определять всю логику обработки запросов как замыкания в файлах маршрутов, вы можете организовать это поведение с помощью классов «контроллеров». Контроллеры могут сгруппировать связанную логику обработки запросов в один класс. Например, класс `UserController` может обрабатывать все входящие запросы, относящиеся к пользователям, включая отображение, создание, обновление и удаление пользователей. По умолчанию контроллеры хранятся в каталоге `app/Http/Controllers`.

## # Написание контроллеров

### Основные понятия о контроллерах

Для быстрого создания нового контроллера вы можете выполнить команду Artisan `make:controller`. По умолчанию все контроллеры для вашего приложения хранятся в каталоге `app/Http/Controllers`:

```
php artisan make:controller UserController
```

Давайте рассмотрим пример базового контроллера. Контроллер может содержать любое количество публичных методов, которые будут отвечать на входящие HTTP-запросы:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать профиль конкретного пользователя.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

После того как вы создали класс контроллера и метод в нем, вы можете определить маршрут к методу контроллера следующим образом:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Когда входящий запрос совпадает с указанным URI маршрута, будет вызван метод `show` класса `App\Http\Controllers\UserController`, и параметры маршрута будут переданы методу.

Контроллеры **не требуют** расширения базового класса. Однако иногда бывает удобно расширить базовый класс контроллера, содержащий методы, которые должны использоваться всеми вашими контроллерами.

## Контроллеры одиночного действия

Если действие контроллера является особенно сложным, вам может показаться удобным посвятить целый класс контроллера этому единственному действию. Для этого вы можете определить один метод `__invoke` в контроллере:

```
<?php

namespace App\Http\Controllers;

class ProvisionServer extends Controller
{
    /**
     * Подготовить новый веб-сервер.
     */
    public function __invoke()
    {
        // ...
    }
}
```

При регистрации маршрутов для контроллеров одиночного действия вам не нужно указывать метод контроллера. Вместо этого вы можете просто передать маршрутизатору имя контроллера:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

Вы можете сгенерировать вызываемый контроллер, используя параметр `--invokable` команды `make:controller` Artisan:

```
php artisan make:controller ProvisionServer --invokable
```

Заготовки контроллера можно настроить с помощью [публикации заготовок](#).

## # Посредник контроллера

[Посредник](#) может быть назначен маршрутам контроллера в ваших файлах маршрутизации:

```
Route::get('/profile', [UserController::class, 'show'])->middleware('auth');
```

Или вам может быть удобно указать посредника в классе контроллера. Для этого ваш контроллер должен реализовать интерфейс [HasMiddleware](#), который требует, чтобы контроллер имел статический метод [middleware](#). Из этого метода вы можете вернуть массив посредников, которые должны быть применены к действиям контроллера:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Routing\Controllers\HasMiddleware;
use Illuminate\Routing\Controllers\Middleware;

class UserController extends Controller implements HasMiddleware
{
    /**
     * Получить посредников, которые должны быть назначены контроллеру.
     */
    public static function middleware(): array
    {
        return [
            'auth',
            new Middleware('log', only: ['index']),
            new Middleware('subscribed', except: ['store']),
        ];
    }
}
```

```
}

// ...

}
```

Вы также можете определить посредника контроллера через замыкание, что обеспечивает удобный способ определения встроенного посредника без написания целого класса посредника:

```
use Closure;
use Illuminate\Http\Request;

/**
 * Get the middleware that should be assigned to the controller.
 */
public static function middleware(): array
{
    return [
        function (Request $request, Closure $next) {
            return $next($request);
        },
    ];
}
```

## # Ресурсные контроллеры

Если вы думаете о каждой модели Eloquent в вашем приложении как о «ресурсе», то для каждого ресурса в вашем приложении обычно выполняются одни и те же наборы действий. Например, представьте, что ваше приложение содержит модель [Photo](#) и модель [Movie](#). Вполне вероятно, что пользователи могут создавать, читать, обновлять или удалять эти ресурсы.

Благодаря такому распространенному варианту использования, маршрутизация ресурсов Laravel присвоит типичные маршруты создания, чтения, обновления и удаления («CRUD») контроллеру с помощью одной строки кода. Для начала мы можем использовать параметр `--resource` команды `make:controller` Artisan, чтобы быстро создать контроллер для обработки этих действий:

```
php artisan make:controller PhotoController --resource
```

Эта команда поместит новый класс контроллера в каталог `app/Http/Controllers` вашего приложения. Контроллер будет содержать метод для каждого из доступных действий с ресурсами. Затем, вы можете зарегистрировать маршрут ресурса, который указывает на контроллер:

```
use App\Http\Controllers\PhotoController;

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

Это единое определение маршрута создаст несколько маршрутов для обработки множества действий с ресурсом. Сгенерированный контроллер уже будет иметь заготовки для каждого из этих действий. Помните, вы всегда можете получить быстрый обзор маршрутов своего приложения, выполнив команду `route:list` Artisan.

Вы даже можете зарегистрировать сразу несколько контроллеров ресурсов, передав массив методу `resources`:

```
Route::resources([
    'photos' => PhotoController::class,
    'posts'  => PostController::class,
]);
```

## Действия, выполняемые ресурсными контроллерами

Метод	URI	Действие	Имя маршрута
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update

Метод	URI	Действие	Имя маршрута
DELETE	/photos/{photo}	destroy	photos.destroy

## Настройка поведения при отсутствии модели

Обычно, если неявно связанная модель ресурса не найдена, то генерируется HTTP-ответ с кодом [404](#). Однако вы можете изменить это поведение, вызвав метод `missing` при определении вашего ресурсного маршрута. Метод `missing` принимает замыкание, которое будет вызываться, если неявно связанная модель не может быть найдена для любого из маршрутов ресурса:

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
    ->missing(function (Request $request) {
        return Redirect::route('photos.index');
});
```

## Модели с мягким удалением

Обычно неявная привязка моделей не будет извлекать модели, которые были [мягко удалены](#), и вместо этого будет возвращать HTTP-ответ 404. Однако вы можете указать фреймворку разрешить использование мягко удаленных моделей, вызвав метод `withTrashed` при определении маршрута ресурса:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->withTrashed();
```

Вызов `withTrashed` без аргументов разрешит использование мягко удаленных моделей для маршрутов ресурса `show`, `edit` и `update`. Вы также можете указать подмножество этих маршрутов, передав массив методу `withTrashed`:

```
Route::resource('photos', PhotoController::class)->withTrashed(['show']);
```

## Указание модели ресурса

Если вы используете [привязку модели к маршруту](#) и хотите, чтобы методы контроллера ресурса содержали типизацию экземпляра модели, вы можете использовать параметр `--model` при создании контроллера:

```
php artisan make:controller PhotoController --model=Photo --resource
```

## Создание запросов формы

Вы можете указать параметр `--requests` при создании контроллера ресурсов, чтобы указать Artisan на создание [классов запросов формы](#) для методов хранения и обновления контроллера:

```
php artisan make:controller PhotoController --model=Photo --resource --requests
```

## Частичные ресурсные маршруты

При объявлении маршрута ресурса вы можете указать подмножество действий, которые должен обрабатывать контроллер, вместо полного набора действий по умолчанию:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

## Ресурсные API-маршруты

При определении маршрутов ресурса, которые будут использоваться API, бывает необходимо исключить маршруты, содержащие ответы с HTML-шаблонами, такие как `create` и `edit`. Для удобства вы можете использовать метод `apiResource`, чтобы автоматически исключить эти два маршрута:

```
use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);
```

Вы можете зарегистрировать сразу несколько ресурсных API-контроллеров, передав массив методу `apiResources`:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\PostController;

Route::apiResources([
    'photos' => PhotoController::class,
    'posts'  => PostController::class,
]);
```

Чтобы быстро сгенерировать ресурсный API-контроллер, который не включает методы `create` или `edit`, используйте переключатель `--api` при выполнении команды `make:controller`:

```
php artisan make:controller PhotoController --api
```

## Вложенные ресурсы

Иногда требуется определить маршруты ко вложенному ресурсу. Например, фоторесурс может иметь несколько комментариев, которые могут быть прикреплены к фотографии. Чтобы вложить ресурсные контроллеры, используйте «точечную нотацию» в определении маршрута:

```
use App\Http\Controllers\PhotoCommentController;

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

Этот маршрут зарегистрирует вложенный ресурс, к которому можно получить доступ с помощью URI, подобных следующим:

```
/photos/{photo}/comments/{comment}
```

## Ограничение вложенных ресурсов

Функционал [неявной привязки модели](#) Laravel может автоматически ограничивать вложенные привязки для подтверждения принадлежности извлеченной дочерней модели по отношению к родительской модели. Используя метод `scoped` при определении вашего вложенного ресурса, вы можете включить автоматическое ограничение, а также указать Laravel, через какое поле дочерний ресурс должен быть получен. Для получения дополнительных сведений о том, как это сделать,смотрите документацию по [ограничению ресурсных маршрутов](#).

## Упрощенное вложение

Часто нет необходимости иметь в URI и родительский, и дочерний идентификаторы, поскольку дочерний идентификатор уже является уникальным идентификатором. При использовании уникальных идентификаторов, таких как автоинкрементные первичные ключи, для идентификации ваших моделей в сегментах URI, вы можете использовать «упрощенное вложение»:

```
use App\Http\Controllers\CommentController;  
  
Route::resource('photos.comments', CommentController::class)->shallow();
```

Это объявление маршрута будет определять следующие маршруты:

Метод	URI	Действие	Имя маршрута
GET	/photos/{photo}/comments	index	photos.comments.index
GET	/photos/{photo}/comments/create	create	photos.comments.create
POST	/photos/{photo}/comments	store	photos.comments.store
GET	/comments/{comment}	show	comments.show
GET	/comments/{comment}/edit	edit	comments.edit
PUT/PATCH	/comments/{comment}	update	comments.update
DELETE	/comments/{comment}	destroy	comments.destroy

# Именование ресурсных маршрутов

По умолчанию все действия ресурсного контроллера имеют имя маршрута; однако, вы можете переопределить эти имена, передав массив имен с желаемыми именами маршрутов:

```
use App\Http\Controllers\PhotoController;

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

# Именование параметров ресурсных маршрутов

По умолчанию `Route::resource` создаст параметры маршрута для ваших ресурсных маршрутов на основе «сингулярной» версии имени ресурса. Вы можете легко переопределить это для каждого ресурса, используя метод `parameters`. Массив, передаваемый в метод `parameters`, должен быть ассоциативным массивом имен ресурсов и имен параметров:

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

В приведенном выше примере создается следующий URI для маршрута `show` ресурса:

```
/users/{admin_user}
```

# Ограничение ресурсных маршрутов

Функционал [ограничения неявной привязки модели](#) Laravel может автоматически ограничивать вложенные привязки для подтверждения принадлежности извлеченной дочерней модели по отношению к родительской модели. Используя метод `scoped` при определении вашего вложенного ресурса, вы можете включить автоматическое ограничение, а также указать Laravel, через какое поле дочерний ресурс должен быть получен:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

Этот маршрут зарегистрирует ограниченный вложенный ресурс, к которому можно получить доступ с помощью таких URI, как следующий:

```
/photos/{photo}/comments/{comment:slug}
```

При использовании пользовательской неявной привязки с ключом в качестве параметра вложенного маршрута, Laravel автоматически задает ограничение для получения вложенной модели своим родителем, используя соглашения, чтобы угадать имя отношения родительского элемента. В этом случае предполагается, что модель `Photo` имеет отношение с именем `comments` (множественное число от имени параметра маршрута), которое можно использовать для получения модели `Comment`.

## Локализация URI ресурсов

По умолчанию `Route::resource` создает URI ресурсов с использованием английских глаголов и правила для множественного числа. Если вам нужно локализовать команды действия `create` и `edit`, вы можете использовать метод `Route::resourceVerbs`. Это можно сделать в начале метода `boot` внутри `App\Providers\AppServiceProvider` вашего приложения:

```
/**
 * Загрузка любых служб приложения.
 */
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

Поддержка множественного числа в Laravel доступна для [нескольких разных языков, которые вы можете настроить в соответствии с вашими потребностями](#). После настройки глаголов и языка множественного числа, регистрация маршрута ресурса, такого как `Route::resource('publicacion', PublicacionController::class)`, будет создавать следующие URI:

`/publicacion/crear`

`/publicacion/{publicacion}/editar`

## Дополнение ресурсных контроллеров

Если вам нужно добавить дополнительные маршруты ресурсного контроллера помимо набора ресурсных маршрутов по умолчанию, вы должны определить эти маршруты перед вызовом метода `Route::resource`; в противном случае маршруты, определенные методом `resource`, могут непреднамеренно иметь приоритет над вашими дополнительными маршрутами:

```
use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

Помните, что ваши контроллеры должны быть сосредоточенными. Если вам постоянно требуются методы, выходящие за рамки типичного набора действий с ресурсами, рассмотрите возможность разделения вашего контроллера на два меньших контроллера.

## Синглтон-ресурс контроллеров

Иногда в вашем приложении могут быть ресурсы, которые могут иметь только один экземпляр. Например, “профиль” пользователя можно редактировать или обновить, но у пользователя может быть только один “профиль”. Точно так же у изображения может быть только одна “миниатюра”. Эти ресурсы называются

“Синглтон-ресурсами” (Singleton resource), что означает, что может существовать только один экземпляр данного ресурса. В таких случаях вы можете зарегистрировать контроллер синглтон-ресурс:

```
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::singleton('profile', ProfileController::class);
```

Вышеописанное определение зарегистрирует следующие маршруты. Как видно, маршруты “создания” не регистрируются для них, и зарегистрированные маршруты не принимают идентификатор, поскольку может существовать только один экземпляр ресурса:

Метод	URI	Действие	Имя маршрута
GET	/profile	show	profile.show
GET	/profile/edit	edit	profile.edit
PUT/PATCH	/profile	update	profile.update

Одиночные ресурсы также могут быть вложены в стандартный ресурс:

```
Route::singleton('photos.thumbnail', ThumbnailController::class);
```

В этом примере ресурс `photos` будет содержать все [стандартные маршруты ресурса](#), однако ресурс `thumbnail` будет синглтон-ресурсом со следующими маршрутами:

Метод	URI	Действие	Имя маршрута
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update

## Создание синглтон-ресурсов

Иногда вам может потребоваться определить маршруты создания и сохранения для синглтон-ресурса. Для этого вы можете вызвать метод `creatable()` при регистрации маршрута синглтон-ресурса:

```
Route::singleton('photos.thumbnail', ThumbnailController::class)->creatable();
```

В этом случае будут зарегистрированы следующие маршруты. Как видно, также будет зарегистрирован маршрут `DELETE` для создаваемых синглтон-ресурсов:

Метод	URI	Действие	Имя маршрута
GET	/photos/{photo}/thumbnail/create	create	photos.thumbnail.create
POST	/photos/{photo}/thumbnail	store	photos.thumbnail.store
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update
DELETE	/photos/{photo}/thumbnail	destroy	photos.thumbnail.destroy

Если вы хотите, чтобы Laravel зарегистрировал маршрут `DELETE` для синглтон-ресурса, но не зарегистрировал маршруты создания или сохранения, вы можете использовать метод `destroyable()`:

```
Route::singleton(...)->destroyable();
```

## Синглтон-ресурсы API

Метод `apiSingleton` можно использовать для регистрации синглтон-ресурса, который будет изменяться через API, и, таким образом, маршруты `create` и `edit` не будут нужны:

```
Route::apiSingleton('profile', ProfileController::class);
```

Конечно же, синглтон-ресурсы API также могут быть `creatable`, что позволит зарегистрировать маршруты `store` и `destroy` для ресурса:

```
Route::apiSingleton('photos.thumbnail', ProfileController::class)->creatable();
```

## # Внедрение зависимостей и контроллеры

### Внедрение зависимостей в конструкторе контроллера

[Контейнер служб](#) Laravel используется для извлечения всех контроллеров. В результате вы можете объявить любые зависимости, которые могут понадобиться вашему контроллеру в его конструкторе. Объявленные зависимости будут автоматически извлечены и внедрены в экземпляр контроллера:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * Создать новый экземпляр контроллера.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}
}
```

### Внедрение зависимостей в методах контроллера

Помимо внедрения в конструкторе, вы также можете объявить тип зависимости в методах вашего контроллера. Распространенный вариант использования внедрения в методе – это внедрение экземпляра `Illuminate\Http\Request` в методы вашего контроллера:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Сохранить нового пользователя.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        //

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

Если ваш метод контроллера также ожидает входные данные из параметра маршрута, укажите аргументы маршрута после других зависимостей. Например, если ваш маршрут определен так:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

Вы по-прежнему можете объявить тип зависимости `Illuminate\Http\Request` и получить доступ к вашему параметру `id`, определив свой метод контроллера следующим образом:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Обновить конкретного пользователя.
     */
```

```
 */
public function update(Request $request, string $id): RedirectResponse
{
    // Обновление пользователя...

    return redirect('/users');
}

}
```

# HTTP-запросы

## # Введение

## # Взаимодействие с запросом

- # Доступ к запросу
- # Path, Host и Method запроса
- # Заголовки запроса
- # IP-адрес запроса
- # Согласование содержимого
- # Запросы стандарта PSR-7

## # Данные полей ввода

- # Получение данных полей ввода
- # Наличие требуемых данных
- # Объединение дополнительных входных данных
- # Данные прошлого запроса
- # Файлы Cookies

## # Обрезание и нормализация значений полей ввода

## # Файлы

- # Получение загруженных файлов
- # Сохранение загруженных файлов

## # Конфигурирование доверенных прокси

## # Конфигурирование доверенных хостов

## # Введение

Класс `Illuminate\Http\Request` Laravel предлагает объектно-ориентированный способ взаимодействия с текущим HTTP-запросом, обрабатываемым вашим приложением, а также извлечение входных данных, файлов Cookies и файлов, отправленных вместе с запросом.

# # Взаимодействие с запросом

## Доступ к запросу

Чтобы получить экземпляр текущего HTTP-запроса через внедрение зависимостей, вы должны объявить класс `Illuminate\Http\Request` в методе контроллера. Экземпляр входящего запроса будет автоматически внедрен [контейнером служб Laravel](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Сохранить нового пользователя.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->input('name');

        // Сохранить пользователя

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

Вы также можете объявить класс `Illuminate\Http\Request` в замыкании маршрута. Контейнер службы автоматически внедрит входящий запрос в замыкание при его выполнении:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

## Внедрение зависимостей и параметры маршрута

Если ваш метод контроллера также ожидает входных данных от параметра маршрута, вы должны указать параметры маршрута после других зависимостей. Например, если ваш маршрут определен так:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

Вы по-прежнему можете объявить `Illuminate\Http\Request` и получить доступ к параметру `id` маршрута, определив метод контроллера так:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Обновить конкретного пользователя.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Обновляем пользователя ...

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

## Path, Host и Method запроса

Экземпляр `Illuminate\Http\Request` содержит множество методов для интерпретации входящего HTTP-запроса и расширяет класс `Symfony\Component\HttpFoundation\Request`. Ниже мы обсудим несколько наиболее важных методов.

### Получение Path запроса

Метод `path` возвращает информацию о пути запроса. Итак, если целевой входящий запрос `http://example.com/foo/bar`, то метод `path` вернет `foo/bar`:

```
$uri = $request->path();
```

## Проверка пути / маршрута запроса

Метод `is` проверит, соответствует ли путь входящего запроса шаблону.

Допускается использование метасимвола подстановки `*`:

```
if ($request->is('admin/*')) {  
    // ...  
}
```

Используя метод `routeIs`, вы можете определить, соответствует ли входящий запрос именованному маршруту:

```
if ($request->routeIs('admin.*')) {  
    // ...  
}
```

## Получение URL-адреса запроса

Чтобы получить полный URL для входящего запроса, вы можете использовать методы `url` или `fullUrl`. Метод `url` вернет URL без строки запроса, а метод `fullUrl`, включая строку запроса:

```
$url = $request->url();  
  
$urlWithQueryString = $request->fullUrl();
```

Если вы хотите добавить данные строки запроса к текущему URL, то вы можете вызвать метод `fullUrlWithQuery`. Этот метод объединяет переданный массив переменных строки запроса с текущей строкой запроса:

```
$request->fullUrlWithQuery(['type' => 'phone']);
```

Если вы хотите получить текущий URL-адрес без заданного параметра строки запроса, вы можете использовать метод `fullUrlWithoutQuery`:

```
$request->fullUrlWithoutQuery(['type']);
```

## Получение хоста(host) запроса

Вы можете получить "host" входящего запроса с помощью методов `host`, `httpHost`, и `schemeAndHttpHost` :

```
$request->host();
$request->httpHost();
$request->schemeAndHttpHost();
```

## Получение метода запроса

Метод `method` вернет HTTP-метод для запроса. Вы можете использовать метод `isMethod` для проверки соответствия HTTP-метода указанной строке:

```
$method = $request->method();

if ($request->isMethod('post')) {
    // ...
}
```

## Заголовки запроса

Вы можете получить заголовок запроса из экземпляра `Illuminate\Http\Request` с помощью метода `header`. Если заголовок отсутствует в запросе, то будетозвращено значение `null`. Однако, метод `header` принимает необязательный второй аргумент, который будет возвращен, если заголовок отсутствует в запросе:

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

Метод `hasHeader` используется, чтобы определить, содержит ли запрос указанный заголовок:

```
if ($request->hasHeader('X-Header-Name')) {  
    // ...  
}
```

Для удобства метод `bearerToken` может использоваться для получения токена из заголовка `Authorization`. Если такого заголовка нет, то будет возвращена пустая строка:

```
$token = $request->bearerToken();
```

## IP-адрес запроса

Метод `ip` используется для получения IP-адреса клиента, который сделал запрос к вашему приложению:

```
$ipAddress = $request->ip();
```

Если вы хотите получить массив IP-адресов, включая все IP-адреса клиентов, которые были перенаправлены прокси-серверами, вы можете использовать метод `ips`. “Исходный” IP-адрес клиента будет находиться в конце массива:

```
$ipAddresses = $request->ips();
```

В общем случае IP-адреса следует считать ненадежным, контролируемым пользователем вводом и использовать их только в информационных целях.

## Согласование содержимого

Laravel содержит несколько методов для проверки типов запрошенного содержимого входящего запроса через заголовок `Accept`. Во-первых, метод `getAcceptableContentTypes` вернет массив, содержащий все типы контента, принятые запросом:

```
$contentTypes = $request->getAcceptableContentTypes();
```

Метод `accepts` принимает массив типов контента и возвращает `true`, если какой-либо из типов контента принят запросом. В противном случае будет возвращено `false`:

```
if ($request->accepts(['text/html', 'application/json'])) {  
    // ...  
}
```

Вы можете использовать метод `prefers`, чтобы определить, какой тип контента, из указанного в массиве типов контента, является наиболее предпочтительным для запроса. Если ни один из предоставленных типов контента не будет принят запросом, будет возвращено значение `null`:

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

Поскольку многие приложения обрабатывают только HTML или JSON, вы можете использовать метод `expectsJson`, чтобы быстро определить, ожидает ли входящий запрос JSON-ответа:

```
if ($request->expectsJson()) {  
    // ...  
}
```

## Запросы стандарта PSR-7

[Стандарт PSR-7](#) определяет интерфейсы для сообщений HTTP, включая запросы и ответы. Если вы хотите получить экземпляр запроса PSR-7 вместо запроса Laravel, вам сначала необходимо установить несколько библиотек. Laravel использует компонент *Symfony HTTP Message Bridge* для преобразования типичных запросов и ответов Laravel в реализации, совместимой с PSR-7:

```
composer require symfony/psr-http-message-bridge  
composer require nyholm/psr7
```

После того как вы установили эти библиотеки, вы можете получить запрос PSR-7, объявив тип интерфейса запроса для замыкания вашего маршрута или контроллера:

```
use Psr\Http\Message\ServerRequestInterface;

Route::get('/', function (ServerRequestInterface $request) {
    // ...
});
```

Если вы возвращаете экземпляр `response` по PSR-7 из маршрута или контроллера, он автоматически преобразуется обратно в экземпляр ответа Laravel и отображается фреймворком.

## # Данные полей ввода

### Получение данных полей ввода

#### Получение данных всех полей ввода

Вы можете получить все данные входящего запроса в виде массива, используя метод `all`. Этот метод можно использовать независимо от того, поступает ли входящий запрос из HTML-формы или является запросом XHR:

```
$input = $request->all();
```

Используя метод `collect`, вы можете получить все входные данные входящего запроса в виде [коллекции](#):

```
$input = $request->collect();
```

Метод `collect` также позволяет вам получить подмножество входных данных входящего запроса в виде коллекции:

```
$request->collect('users')->each(function (string $user) {
    // ...
```

```
});
```

## Получение значения конкретного поля ввода

Используя несколько простых методов, вы можете получить доступ ко всем поступившим от пользователя данным, используя экземпляр `Illuminate\Http\Request`, не беспокоясь о том, какой HTTP-метод использовался для запроса. Независимо от HTTP-метода, для получения этих данных может использоваться метод `input`:

```
$name = $request->input('name');
```

Вы можете передать значение по умолчанию в качестве второго аргумента метода `input`. Это значение будет возвращено, если запрошенное значение отсутствует в запросе:

```
$name = $request->input('name', 'Sally');
```

При работе с формами, содержащими массив входных данных, используйте «точечную» нотацию для доступа к элементам массива:

```
$name = $request->input('products.0.name');
```

```
$names = $request->input('products.*.name');
```

Вы можете вызвать метод `input` без аргументов, чтобы получить все значения входных данных в виде ассоциативного массива:

```
$input = $request->input();
```

## Получение данных из строки запроса

В то время как метод `input` извлекает значения из всей информационной части данных запроса (включая строку запроса), метод `query` извлекает значения только из строки запроса:

```
$name = $request->query('name');
```

Если значение данных из строки запроса отсутствуют, будет возвращен второй аргумент этого метода:

```
$name = $request->query('name', 'Helen');
```

Вы можете вызвать метод `query` без аргументов, чтобы получить все значения строки запроса в виде ассоциативного массива:

```
$query = $request->query();
```

## Получение значений JSON-содержимого

При отправке запросов JSON в ваше приложение, вы можете получить доступ к данным JSON с помощью метода `input`, если заголовок запроса `Content-Type` корректно установлен как `application/json`. Вы даже можете использовать «точечную» нотацию для извлечения значений, вложенных в JSON-массивы или объекты:

```
$name = $request->input('user.name');
```

## Получение экземпляра `Stringable` из `Input`

Вместо получения входных данных запроса в виде примитивной `string` вы можете использовать метод `string` для получения данных запроса как экземпляра [`Illuminate\Support\Stringable`](#):

```
$name = $request->string('name')->trim();
```

## Получение целочисленных входных значений

Чтобы получить входные значения в виде целых чисел, вы можете использовать метод `integer`. Этот метод попытается привести входное значение к целому числу. Если входные данные отсутствуют или приведение не удалось, оно вернет

указанное вами значение по умолчанию. Это особенно полезно для нумерации страниц или других числовых входных данных:

```
$perPage = $request->integer('per_page');
```

## Получение значений логического типа

При работе с элементами HTML, такими как флаги, ваше приложение может получать «логические» значения, которые на самом деле являются строками. Например, строковые «true» или «on». Для удобства вы можете использовать метод `boolean`, чтобы получить эти значения как логические. Метод `boolean` возвращает `true` для `1`, `true`, и строковых «`1`», «`true`», «`on`» и «`yes`». Все остальные значения вернут `false`:

```
$archived = $request->boolean('archived');
```

## Получение значений Даты

Для удобства входные значения, содержащие дату/время, могут быть получены как экземпляры Carbon с использованием метода `date`. Если запрос не содержит входного значения с заданным именем, будет возвращен `null`:

```
$birthday = $request->date('birthday');
```

Второй и третий аргументы, принятые методом `date`, могут использоваться для указания формата даты и часового пояса соответственно:

```
$elapsed = $request->date('elapsed', '!H:i', 'Europe/Madrid');
```

Если входное значение присутствует, но имеет недопустимый формат, будет выброшено исключение `InvalidArgumentException`, поэтому рекомендуется проверять ввод перед вызовом метода `date`.

## Retrieving Enum Input Values

Входные значения, соответствующие [PHP enums](#), также могут быть извлечены из запроса. Если запрос не содержит входного значения с заданным именем или

`enum` не имеет значения, соответствующего входному значению из `request`, будет возвращен `null`. Метод `enum` принимает имя входного значения из `request` первым аргументом и класс с перечислениями `enums` в качестве второго:

```
use App\Enums>Status;  
  
$status = $request->enum('status', Status::class);
```

## Получение данных через динамические свойства

Вы также можете получить доступ к поступившим от пользователя данным, используя динамические свойства экземпляра `Illuminate\Http\Request`. Например, если одна из форм вашего приложения содержит поле `name`, то вы можете получить доступ к значению поля следующим образом:

```
$name = $request->name;
```

При использовании динамических свойств Laravel сначала будет искать значение параметра в информационной части данных запроса. Если его нет, Laravel будет искать поле в соответствующих параметрах маршрута.

## Частичное получение данных полей ввода

Если вам нужно получить подмножество входных данных, вы можете использовать методы `only` и `except`. Оба метода принимают один массив или динамический список аргументов:

```
$input = $request->only(['username', 'password']);  
  
$input = $request->only('username', 'password');  
  
$input = $request->except(['credit_card']);  
  
$input = $request->except('credit_card');
```

Метод `only` возвращает все запрошенные вами пары ключ/значение; однако он не будет возвращать

пары ключ/значение, которых нет в запросе.

## Наличие требуемых данных

Вы можете использовать метод `has`, чтобы определить, присутствует ли значение в запросе. Метод `has` возвращает `true`, если значение присутствует в запросе:

```
if ($request->has('name')) {  
    // ...  
}
```

При передаче массива метод `has` определяет, присутствуют ли все указанные значения:

```
if ($request->has(['name', 'email'])) {  
    // ...  
}
```

Метод `hasAny` возвращает `true`, если присутствует любое из указанных значений:

```
if ($request->hasAny(['name', 'email'])) {  
    // ...  
}
```

Метод `whenHas` выполнит переданное замыкание, если в запросе присутствует значение:

```
$request->whenHas('name', function (string $input) {  
    // ...  
});
```

Второе замыкание может быть передано методу `whenHas`, которое будет выполнено, если указанное значение отсутствует в запросе:

```
$request->whenHas('name', function (string $input) {  
    // Значение "имя" присутствует...
```

```
}, function () {
    // Значение "имя" отсутствует...
});
```

Если вы хотите определить, присутствует ли значение в запросе и не является ли оно пустой строкой, вы можете использовать метод `filled`:

```
if ($request->filled('name')) {
    // ...
}
```

Если вы хотите определить, отсутствует ли значение в запросе или является ли оно пустой строкой, вы можете использовать метод `isNotFilled`:

```
if ($request->isNotFilled('name')) {
    // ...
}
```

При получении массива метод `isNotFilled` определит, все ли указанные значения отсутствуют или пусты:

```
if ($request->isNotFilled(['name', 'email'])) {
    // ...
}
```

Метод `anyFilled` возвращает `true`, если какое-либо из указанных значений не является пустой строкой:

```
if ($request->anyFilled(['name', 'email'])) {
    // ...
}}
```

Метод `whenFilled` выполнит указанное замыкание, если значение присутствует в запросе и не является пустой строкой:

```
$request->whenFilled('name', function (string $input) {
    // ...
```

```
});
```

Второе замыкание может быть передано методу `whenFilled` которое будет выполнено, если указанное значение «не заполнено»:

```
$request->whenFilled('name', function ($input) {
    // Значение "имя" заполнено ...
}, function () {
    // Значение "имя" не заполнено...
});
```

Чтобы определить, отсутствует ли конкретный ключ в запросе, вы можете использовать метод `missing` или `whenMissing`:

```
if ($request->missing('name')) {
    // ...
}

$request->whenMissing('name', function () {
    // Значение "name" пропущено...
}, function () {
    // Значение "name" присутствует...
});
```

## Объединение дополнительных входных данных

Иногда вам может потребоваться вручную объединить дополнительные входные данные с существующими входными данными запроса. Для достижения этой цели вы можете использовать метод `merge`. Если данный входной ключ уже существует в запросе, он будет перезаписан данными, предоставленными методу `merge`:

```
$request->merge(['votes' => 0]);
```

Метод `mergeIfMissing` может использоваться для объединения ввода с запросом, если соответствующие ключи еще не существуют во входных данных запроса:

```
$request->mergeIfMissing(['votes' => 0]);
```

# Данные прошлого запроса

Laravel позволяет вам сохранить входные данные из текущего запроса на время выполнения следующего запроса. Эта функция особенно полезна для повторного заполнения форм после обнаружения ошибок валидации. Однако, если вы используете содержащуюся в Laravel [валидацию](#), то, возможно, вам не придется вручную использовать эти методы кратковременного сохранения входных данных в сессии напрямую, так как некоторые встроенные средства валидации Laravel будут вызывать их автоматически.

## Кратковременное сохранение входных данных в сессии

Метод `flash` класса `Illuminate\Http\Request` будет сохранять входные данные в [сессии](#), чтобы они были доступны **только** во время следующего запроса пользователя к приложению:

```
$request->flash();
```

Вы также можете использовать методы `flashOnly` и `flashExcept` для передачи подмножества данных запроса в сессию. Эти методы полезны для скрытия конфиденциальной информации из сессии, например, пароли:

```
$request->flashOnly(['username', 'email']);  
$request->flashExcept('password');
```

## Кратковременное сохранение при перенаправлении

Так как вам часто нужно выполнять кратковременное сохранение входных данных в сессии, а затем перенаправлять на предыдущую страницу, вы можете легко связать сохранение данных с перенаправлением, используя метод `withInput`:

```
return redirect('/form')->withInput();  
  
return redirect()->route('user.create')->withInput();  
  
return redirect('/form')->withInput(  
    $request->except('password')  
)
```

## Получение данных прошлого запроса

Чтобы получить кратковременно сохраненные входные данные из предыдущего запроса, вызовите метод `old` экземпляра `Illuminate\Http\Request`. Метод `old` извлечет ранее записанные входные данные из [сессии](#):

```
$username = $request->old('username');
```

Laravel также содержит глобального помощника `old`. Если вы показываете данные из предыдущего запроса в [шаблоне Blade](#), удобнее использовать помощник `old` для повторного заполнения формы. Если для поля не были указаны данные в предыдущем запросе, то будет возвращен `null`:

```
<input type="text" name="username" value="{{ old('username') }}">
```

## Файлы Cookies

### Получение файлов Cookies из запроса

Все файлы Cookies, созданные фреймворком Laravel, зашифрованы и подписаны кодом аутентификации, что означает, что они будут считаться недействительными, если они были изменены клиентом. Чтобы получить значение cookie из запроса, используйте метод `cookie` экземпляра `Illuminate\Http\Request`:

```
$value = $request->cookie('name');
```

## # Обрезание и нормализация значений полей ввода

По умолчанию Laravel содержит посредников `Illuminate\Foundation\Http\Middleware\TrimStrings` и `Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull` в глобальном стеке посредников вашего приложения. Первый из упомянутых посредников будет автоматически обрезать все входящие строковые поля запроса, а второй – конвертировать любые пустые строковые поля в `null`. Это позволяет вам

не беспокоиться об этих проблемах нормализации в ваших маршрутах и контроллерах.

## Отключение нормализации значений полей ввода

Если вы хотите отключить это поведение для всех запросов, вы можете удалить два посредника из стека посредников вашего приложения, вызвав метод `$middleware->remove` в файле `bootstrap/app.php` вашего приложения:

```
use Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull;
use Illuminate\Foundation\Http\Middleware\TrimStrings;

->withMiddleware(function (Middleware $middleware) {
    $middleware->remove([
        ConvertEmptyStringsToNull::class,
        TrimStrings::class,
    ]);
})
```

Если вы хотите отключить обрезку строк и преобразование пустых строк для подмножества запросов к вашему приложению, вы можете использовать методы посредника `trimStrings` и `convertEmptyStringsToNull` в файле `bootstrap/app.php` вашего приложения. Оба метода принимают массив замыканий, который должен возвращать `true` или `false`, чтобы указать, следует ли пропустить нормализацию ввода:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->convertEmptyStringsToNull(except: [
        fn (Request $request) => $request->is('admin/*'),
    ]);

    $middleware->trimStrings(except: [
        fn (Request $request) => $request->is('admin/*'),
    ]);
})
```

## # Файлы

### Получение загруженных файлов

Вы можете получить загруженные файлы из экземпляра `Illuminate\Http\Request`, используя метод `file` или динамические свойства. Метод `file` возвращает экземпляр класса `Illuminate\Http\UploadedFile`, который расширяет класс `SplFileInfo` PHP и содержит различные методы для взаимодействия с файлом:

```
$file = $request->file('photo');

$file = $request->photo;
```

Вы можете определить, представлен ли файл в запросе, используя метод `hasFile`:

```
if ($request->hasFile('photo')) {
    // ...
}
```

## Валидация загрузки файлов

Помимо проверки наличия файла, вы можете убедиться, что не было ли каких-либо проблем с загрузкой файла с помощью метода `isValid`:

```
if ($request->file('photo')->isValid()) {
    // ...
}
```

## Пути к файлам и расширения

Класс `UploadedFile` также содержит методы для доступа к полному пути файла и его расширению. Метод `extension` попытается угадать расширение файла на основе его содержимого. Это расширение может отличаться от расширения, предоставленного клиентом:

```
$path = $request->photo->path();

$extension = $request->photo->extension();
```

## Другие методы для работы с загружаемыми файлами

Для экземпляров `UploadedFile` доступно множество других методов.

Дополнительные сведения об этих методах смотрите в [документации по API](#) для этого класса.

## Сохранение загруженных файлов

Чтобы сохранить загруженный файл, вы обычно будете использовать одно из ваших настроенных [файловых хранилищ](#). Класс `UploadedFile` имеет метод `store`, который перемещает загруженный файл на один из ваших дисков, находящийся в вашей локальной файловой системе или в облачном хранилище, таком как Amazon S3.

Метод `store` принимает путь, по которому файл должен храниться относительно настроенного корневого каталога файловой системы. Этот путь не должен содержать имени файла, поскольку в качестве имени файла будет автоматически создан уникальный идентификатор.

Метод `store` также принимает необязательный второй аргумент для имени диска, который следует использовать для хранения файла. Метод вернет путь к файлу относительно корня диска:

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

Если вы не хотите, чтобы имя файла создавалось автоматически, вы можете использовать метод `storeAs`, который принимает в качестве аргументов путь, имя файла и имя диска:

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

Для получения дополнительной информации о хранилище файлов в Laravel, ознакомьтесь с полной [документацией по файловому хранилищу](#).

# # Конфигурирование доверенных прокси

При запуске ваших приложений, использующих балансировщик нагрузки, завершающий сертификаты TLS / SSL, вы можете заметить, что ваше приложение иногда не генерирует ссылки протокола HTTPS при использовании глобального помощника [url](#). Обычно это связано с тем, что ваше приложение перенаправляет трафик от вашего балансировщика нагрузки на порт 80 и не знает, что оно должно генерировать безопасные ссылки.

Чтобы решить эту проблему, вы можете использовать посредника [Illuminate\Http\Middleware\TrustProxies](#), содержащийся в вашем приложении Laravel, что позволяет вам быстро настраивать балансировщики нагрузки или прокси, которым ваше приложение должно доверять. Доверенные прокси-серверы должны быть указаны с помощью метода посредника [trustProxies](#) в файле [bootstrap/app.php](#) вашего приложения:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(at: [
        '192.168.1.1',
        '10.0.0.0/8',
    ]);
})
```

Помимо настройки доверенных прокси-серверов, вы также можете настроить заголовки прокси-серверов, которым следует доверять:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(headers: Request::HEADER_X_FORWARDED_FOR | Request::HEADER_X_FORWARDED_HOST | Request::HEADER_X_FORWARDED_PORT | Request::HEADER_X_FORWARDED_PROTO | Request::HEADER_X_FORWARDED_AWS_ELB);
})
```

Если вы используете AWS Elastic Load Balancing, значение [headers](#) должно быть [Request::HEADER\\_X\\_FORWARDED\\_AWS\\_ELB](#). Если ваш балансировщик нагрузки использует стандартный

заголовок `Forwarded` из [RFC 7239] (<https://www.rfc-editor.org/rfc/rfc7239#section-4>), значение `headers` должно быть `Request::HEADER_FORWARDED`. Для получения дополнительной информации о константах, которые можно использовать в значении `headers`, ознакомьтесь с документацией Symfony о [доверенных прокси-серверах] (<https://symfony.com/doc/7.0/deployment/proxies.html>).

## Доверие ко всем прокси

Если вы используете Amazon AWS или другой поставщик «облачных» балансировщиков нагрузки, то вы можете не знать IP-адреса своих фактических балансировщиков. В этом случае вы можете использовать `*`, чтобы доверять всем прокси:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(at: '*');
})
```

## # Конфигурирование доверенных хостов

По умолчанию Laravel будет отвечать на все запросы, которые он получает, независимо от содержимого заголовка `Host` HTTP-запроса. Кроме того, значение заголовка `Host` будет использоваться при генерации абсолютных URL-адресов вашего приложения во время веб-запроса.

Как правило, вам следует настроить свой веб-сервер (Nginx или Apache), так, чтобы он обслуживал запросы, соответствующие только указанному имени хоста. Однако, если у вас нет возможности напрямую настроить свой веб-сервер и вам нужно указать Laravel, чтобы он отвечал только на определенные имена хостов, вы можете сделать это, задействовав посредник `Illuminate\Http\Middleware\TrustHosts` для вашего приложения.

Чтобы включить посредника `TrustHosts`, вам следует вызвать метод посредника `trustHosts` в файле `bootstrap/app.php` вашего приложения. Используя аргумент `at`

этого метода, вы можете указать имена хостов, на которые ваше приложение должно реагировать. Входящие запросы с другими заголовками `Host` будут отклонены:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustHosts(at: ['laravel.test']);
})
```

По умолчанию запросы, поступающие из поддоменов URL-адреса приложения, также автоматически считаются доверенными. Если вы хотите отключить это поведение, вы можете использовать аргумент `subdomains`:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustHosts(at: ['laravel.test'], subdomains: false);
})
```

Если вам нужен доступ к файлам конфигурации или базе данных вашего приложения, чтобы определить доверенные хосты, вы можете предоставить замыкание аргументу `at`:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustHosts(at: fn () => config('app.trusted_hosts'));
})
```

# HTTP-ответы

## # Создание ответов

- # Добавление заголовков к ответам
- # Добавление файлов Cookies к ответам
- # Файлы Cookies и шифрование

## # Перенаправления

- # Перенаправление на именованные маршруты
- # Перенаправление к действиям контроллера
- # Перенаправление на внешние домены
- # Перенаправление с кратковременным сохранением данных в сессии

## # Другие типы ответов

- # Ответы с HTML-шаблонами
- # Ответы JSON
- # Ответы для загрузки файлов
- # Ответы на файлы
- # Потоковые ответы

## # Макрокоманды ответа

## # Создание ответов

### Строки и массивы

Все маршруты и контроллеры должны возвращать ответ, который будет отправлен обратно в браузер пользователя. Laravel предлагает несколько разных способов вернуть ответы. Самый простой ответ – это возврат строки из маршрута или контроллера. Фреймворк автоматически преобразует строку в полный HTTP-ответ:

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

Помимо возврата строк из ваших маршрутов и контроллеров, вы также можете возвращать массивы. Фреймворк автоматически преобразует массив в ответ JSON:

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

Знаете ли вы, что можете возвращать [коллекции Eloquent](#) из ваших маршрутов или контроллеров? Они будут автоматически преобразованы в JSON.

## Объекты ответа

Как правило, вы не просто будете возвращать строки или массивы из действий маршрута. Вместо этого вы вернете полные экземпляры [Illuminate\Http\Response](#) или [шаблоны](#).

Возврат полного экземпляра [Response](#) позволяет вам настроить код состояния и заголовки HTTP ответа. Экземпляр [Response](#) наследуется от класса [Symfony\Component\HttpFoundation\Response](#), который содержит множество методов для построения ответов HTTP:

```
Route::get('/home', function () {
    return response('Hello World', 200)
        ->header('Content-Type', 'text/plain');
});
```

## Модели и коллекции Eloquent

По желанию можно вернуть модели и коллекции [Eloquent ORM](#) прямо из ваших маршрутов и контроллеров. Когда вы это сделаете, Laravel автоматически преобразует модели и коллекции в ответы JSON, учитывая [скрытие атрибутов](#) модели:

```
use App\Models\User;
```

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

## Добавление заголовков к ответам

Имейте в виду, что большинство методов ответа можно объединять в цепочку вызовов для гибкого создания экземпляров ответа. Например, вы можете использовать метод `header` для добавления серии заголовков к ответу перед его отправкой обратно пользователю:

```
return response($content)
    ->header('Content-Type', $type)
    ->header('X-Header-One', 'Header Value')
    ->header('X-Header-Two', 'Header Value');
```

Или вы можете использовать метод `withHeaders`, чтобы указать массив заголовков, которые будут добавлены к ответу:

```
return response($content)
    ->withHeaders([
        'Content-Type' => $type,
        'X-Header-One' => 'Header Value',
        'X-Header-Two' => 'Header Value',
    ]);
```

## Посредник управления кешем

Laravel содержит посредник `cache.headers`, используемый для быстрой установки заголовка `Cache-Control` для группы маршрутов. Директивы должны быть предоставлены с использованием эквивалента "snake case" соответствующей директивы управления кешем и должны быть разделены точкой с запятой. Если в списке директив указан `etag`, то MD5-хеш содержимого ответа будет автоматически установлен как идентификатор ETag:

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(function () {
    Route::get('/privacy', function () {
        // ...
    });
});
```

```
Route::get('/terms', function () {
    // ...
});
});
```

## Добавление файлов Cookies к ответам

Вы можете добавить Cookies к исходящему экземпляру `Illuminate\Http\Response`, используя метод `cookie`. Вы должны передать этому методу имя, значение и количество минут, в течение которых куки должен считаться действительным:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

Метод `cookie` также принимает еще несколько аргументов, которые используются реже. Как правило, эти аргументы имеют то же назначение и значение, что и аргументы, передаваемые встроенному в PHP методу `setcookie` method:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

Если вы хотите, чтобы куки отправлялся вместе с исходящим ответом, но у вас еще нет экземпляра этого ответа, вы можете использовать фасад `Cookie`, чтобы «поставить в очередь» файлы Cookies для добавления их к ответу при его отправке. Метод `queue` принимает аргументы, необходимые для создания экземпляра `Cookie`. Эти файлы Cookies будут добавлены к исходящему ответу перед его отправкой в браузер:

```
use Illuminate\Support\Facades\Cookie;

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

## Создание экземпляров Cookie

Если вы хотите сгенерировать экземпляр `Symfony\Component\HttpFoundation\Cookie`, который может быть добавлен к экземпляру ответа позже, вы можете

использовать глобальный помощник `cookie`. Этот файл Cookies не будет отправлен обратно клиенту, если он не прикреплен к экземпляру ответа:

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

## Досрочное окончание срока действия файлов Cookies

Вы можете удалить куки, обнулив срок его действия с помощью метода `withoutCookie` исходящего ответа:

```
return response('Hello World')->withoutCookie('name');
```

Если у вас еще нет экземпляра исходящего ответа, вы можете использовать метод `expire` фасада `Cookie` для обнуления срока действия кук:

```
Cookie::expire('name');
```

## Файлы Cookies и шифрование

По умолчанию, благодаря middleware `Illuminate\Cookie\Middleware\EncryptCookies` все файлы Cookies, генерируемые Laravel, зашифрованы и подписаны, поэтому клиент не может их изменить или прочитать. Если вы хотите отключить шифрование для подмножества куки, созданных вашим приложением, вы можете использовать метод `encryptCookies` в файле `bootstrap/app.php` вашего приложения:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->encryptCookies(except: [
        'cookie_name',
    ]);
})
```

## # Перенаправления

Ответы с перенаправлением являются экземплярами класса `Illuminate\Http\RedirectResponse` и содержат корректные заголовки, необходимые

для перенаправления пользователя на другой URL. Есть несколько способов сгенерировать экземпляр `RedirectResponse`. Самый простой способ – использовать глобальный помощник `redirect`:

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

По желанию можно перенаправить пользователя в его предыдущее местоположение, например, когда отправленная форма является недействительной. Вы можете сделать это с помощью глобального помощника `back`. Поскольку эта функция использует `сессии`, убедитесь, что маршрут, вызывающий функцию `back`, использует группу посредников `web`:

```
Route::post('/user/profile', function () {
    // Валидация запроса ...

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

## Перенаправление на именованные маршруты

Когда вы вызываете помощник `redirect` без параметров, возвращается экземпляр `Illuminate\Routing\Redirector`, что позволяет вам вызывать любой метод экземпляра `Redirector`. Например, чтобы сгенерировать `RedirectResponse` на именованный маршрут, вы можете использовать метод `route`:

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

Если ваш маршрут имеет параметры, вы можете передать их в качестве второго аргумента методу `route`:

```
// Для маршрута со следующим URI: /profile/{id}

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

## Заполнение параметров с моделей Eloquent

Если вы перенаправляете на маршрут с параметром `ID`, который извлекается из модели Eloquent, то вы можете просто передать саму модель. ID будет извлечен автоматически:

```
// Для маршрута со следующим URI: /profile/{id}

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

Если вы хотите настроить значение, которое соответствует параметру маршрута, то вы можете указать столбец при определении параметра маршрута (`/profile/{id:slug}`) или переопределить метод `getRouteKey` в вашей модели Eloquent:

```
/**
 * Получить значение ключа маршрута модели.
 */
public function getRouteKey(): mixed
{
    return $this->slug;
}
```

## Перенаправление к действиям контроллера

Вы также можете генерировать перенаправления на [действия контроллера](#). Для этого передайте имя контроллера и действия методу `action`:

```
use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);
```

Если ваш маршрут контроллера требует параметров, вы можете передать их в качестве второго аргумента методу `action`:

```
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

## Перенаправление на внешние домены

Иногда может потребоваться перенаправление на домен за пределами вашего приложения. Вы можете сделать это, вызвав метод `away`, который создает `RedirectResponse` без какой-либо дополнительной кодировки URL, валидации или проверки:

```
return redirect()->away('https://www.google.com');
```

## Перенаправление с кратковременным сохранением данных в сессии

Перенаправление на новый URL-адрес и [краткосрочная запись данных в сессию](#) обычно выполняются одновременно. Обычно это делается после успешного выполнения действия, когда вы отправляете сообщение об успешном завершении в сессию. Для удобства вы можете создать экземпляр `RedirectResponse` и передать данные в сессию в единой текучей цепочке методов:

```
Route::post('/user/profile', function () {
    // ...

    return redirect('/dashboard')->with('status', 'Profile updated!');
});
```

После перенаправления пользователя, вы можете отобразить сохраненное из [сессии](#) сообщение. Например, используя [синтаксис Blade](#):

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

## Перенаправление с кратковременным сохранением входных данных

Вы можете использовать метод `withInput` экземпляра `RedirectResponse`, для передачи входных данных текущего запроса в сессию перед перенаправлением пользователя в новое место. Обычно это делается, если пользователь спровоцировал ошибку валидации. После того как входные данные были переданы

в сессию, вы можете легко [получить их](#) во время следующего запроса для повторного автозаполнения формы:

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

## # Другие типы ответов

Помощник `response` используется для генерации других типов экземпляров ответа. Когда помощник `response` вызывается без аргументов, возвращается реализация [контракта Illuminate\Contracts\Routing\ResponseFactory](#). Этот контракт содержит несколько полезных методов для генерации ответов.

### Ответы с HTML-шаблонами

Если вам нужен контроль над статусом и заголовками ответа, но также необходимо вернуть [HTML-шаблон](#) в качестве содержимого ответа, то вы должны использовать метод `view`:

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

Конечно, вы можете использовать глобальный помощник `view`, даже если вам не нужно передавать собственные код состояния или заголовки HTTP.

### Ответы JSON

Метод `json` автоматически установит заголовок `Content-Type` в `application/json`, а также преобразует переданный массив в JSON с помощью функции `json_encode` PHP:

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

Если вы хотите создать ответ JSONP, вы можете использовать метод `json` в сочетании с методом `withCallback`:

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

## Ответы для загрузки файлов

Метод `download` используется для генерации ответа, который заставляет браузер пользователя загружать файл по указанному пути. Метод `download` принимает имя файла в качестве второго аргумента метода, определяющий имя файла, которое видит пользователь, загружающий файл. Наконец, вы можете передать массив заголовков HTTP в качестве третьего аргумента метода:

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

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

Symfony HttpFoundation, управляющий загрузкой файлов, требует, чтобы имя загружаемого файла было в кодировке ASCII.

## Ответы на файлы

Метод `file` может использоваться для отображения файла, например изображения или PDF-файла, непосредственно в браузере пользователя вместо запуска загрузки. Этот метод принимает абсолютный путь к файлу в качестве первого аргумента и массив заголовков в качестве второго аргумента:

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

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

## Потоковые ответы

Передавая данные клиенту по мере их создания, вы можете значительно сократить использование памяти и повысить производительность, особенно для очень больших ответов. Потоковые ответы позволяют клиенту начать обработку данных до того, как сервер завершит их отправку:

```
function streamedContent(): Generator {
    yield 'Hello, ';
    yield 'World!';
}

Route::get('/stream', function () {
    return response()->stream(function (): void {
        foreach (streamedContent() as $chunk) {
            echo $chunk;
            ob_flush();
            flush();
            sleep(2); // Simulate delay between chunks...
        }
    }, 200, ['X-Accel-Buffering' => 'no']);
});
```

Внутри Laravel использует функцию буферизации вывода PHP. Как вы можете видеть в приведенном выше примере, вам следует использовать функции `ob_flush` и `flush` для отправки буферизованного содержимого клиенту.

## Потоковые ответы JSON

Если вам нужно поэтапно передавать данные JSON, вы можете использовать метод `streamJson`. Этот метод особенно полезен для больших наборов данных, которые необходимо постепенно отправлять в браузер в формате, который можно легко проанализировать с помощью JavaScript:

```
use App\Models\User;

Route::get('/users.json', function () {
    return response()->streamJson([
        'users' => User::cursor(),
```

```
]);  
});
```

## Потоковые загрузки

По желанию можно превратить строковый ответ переданной функции в загружаемый ответ без необходимости записывать результирующее содержимое на диск. В этом сценарии вы можете использовать метод `streamDownload`. Этот метод принимает в качестве аргументов замыкание, имя файла и необязательный массив заголовков:

```
use App\Services\GitHub;  
  
return response()->streamDownload(function () {  
    echo GitHub::api('repo')  
        ->contents()  
        ->readme('laravel', 'laravel')['contents'];  
}, 'laravel-readme');
```

## # Макрокоманды ответа

Если вы хотите определить собственный ответ, который вы можете повторно использовать в различных маршрутах и контроллерах, то вы можете использовать метод `macro` фасада `Response`. Как правило, этот метод следует вызывать в методе `boot` одного из [поставщиков служб](#) вашего приложения, например, `App\Providers\AppServiceProvider`:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Response;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Загрузка любых служб приложения.  
     */  
    public function boot(): void  
    {  
        Response::macro('caps', function (string $value) {
```

```
        return Response::make(strtoupper($value));
    });
}
}
```

Метод `macro` принимает имя как свой первый аргумент и замыкание – как второй аргумент. Замыкание макрокоманды будет выполнено при вызове имени макрокоманды из реализации `ResponseFactory` или глобального помощника `response`:

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

# HTML-шаблоны

## # Введение

# Шаблоны React / Vue

## # Создание и отрисовка шаблонов

# Вложенные каталоги шаблонов

# Использование первого доступного шаблона

# Определение наличия шаблона

## # Передача данных шаблону

# Общедоступные данные для всех шаблонов

## # Компоновщики шаблонов

# Создатели шаблонов

## # Оптимизация шаблонов

## # Введение

Конечно, не практично возвращать целые строки HTML напрямую из ваших маршрутов и контроллеров. К счастью, в Laravel есть представления (views), которые предоставляют удобный способ размещения всего HTML в отдельных файлах.

Представления (views) разделяют вашу логику контроллера или приложения от логики представления и хранятся в каталоге `resources/views`. При использовании Laravel обычно шаблоны представлений написаны с использованием [языка шаблонизации Blade](#). Простое представление может выглядеть примерно так:

```
<!-- Шаблон сохранен в `resources/views/greeting.blade.php` -->

<html>
  <body>
    <h1>Привет, {{ $name }}</h1>
  </body>
</html>
```

Поскольку этот шаблон сохранен в `resources/views/greeting.blade.php`, мы можем вернуть его, используя глобальный помощник `view`, например:

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

Ищете дополнительную информацию о том, как писать шаблоны Blade? Ознакомьтесь с полной [документацией по Blade](#), чтобы начать работу.

## Шаблоны React / Vue

Вместо написания фронтенд-шаблонов на PHP с использованием Blade многие разработчики предпочитают писать свои шаблоны с использованием React или Vue. Laravel делает это легким благодаря [Inertia](#), библиотеке, которая упрощает связь вашего фронтенда на React или Vue с вашим бэкендом Laravel, избегая типичных сложностей, связанных с созданием SPA (Single Page Application).

Наши стартовые наборы Breeze и Jetstream [starter kits](#) предоставляют отличную отправную точку для вашего следующего приложения Laravel, работающего на Inertia. Кроме того, [Laravel Bootcamp](#) предоставляет полное демонстрационное руководство по созданию приложения Laravel, работающего на Inertia, включая примеры с использованием Vue и React.

## # Создание и отрисовка шаблонов

Вы можете создать представление, разместив файл с расширением `.blade.php` в каталоге `resources/views` вашего приложения, либо с помощью команды Artisan `make:view`:

```
php artisan make:view greeting
```

Расширение `.blade.php` информирует фреймворк о том, что файл является [шаблоном Blade](#). Blade-шаблоны содержат HTML, а также директивы Blade,

которые позволяют вам легко выводить значения, создавать условные операторы "if", выполнять итерации по данным и многое другое.

После того как вы создали шаблон, вы можете вернуть его из маршрута или контроллера вашего приложения, используя глобальный помощник `view`:

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

Шаблон также могут быть возвращены с помощью фасада `View`:

```
use Illuminate\Support\Facades\View;

return View::make('greeting', ['name' => 'James']);
```

Как видно, первый аргумент, переданный помощнику `view`, соответствует имени файла шаблона в каталоге `resources/views`. Второй аргумент – это массив данных, которые должны быть доступны в шаблоне. В этом случае мы передаем переменную `name`, которая будет выведена в шаблоне с использованием [синтаксиса Blade](#).

## Вложенные каталоги шаблонов

Шаблоны также могут быть вложены в подкаталоги каталога `resources/views`. «Точечная нотация» используется для указания вложенности шаблона. Например, если ваш шаблон хранится в `resources/views/admin/profile.blade.php`, то вы можете вернуть его из маршрута / контроллера вашего приложения следующим образом:

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

Имена каталогов шаблонов не должны содержать символа ..

## Использование первого доступного шаблона

Используя метод `first` фасада `View`, вы можете отобразить первый шаблон, который существует в переданном массиве шаблонов. Это может быть полезно, если ваше приложение или пакет позволяют настраивать или перезаписывать шаблоны:

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

## Определение наличия шаблона

Если вам нужно определить, существует ли шаблон, вы можете использовать фасад `View`. Метод `exists` вернет `true`, если он существует:

```
use Illuminate\Support\Facades\View;

if (View::exists('admin.profile')) {
    // ...
}
```

## # Передача данных шаблону

Как вы видели в предыдущих примерах, вы можете передать массив данных шаблонам, чтобы сделать эти данные доступными для них:

```
return view('greetings', ['name' => 'Victoria']);
```

При передаче информации таким образом данные должны быть массивом с парами ключ / значение. После предоставления данных в шаблон вы можете получить доступ к каждому значению, используя ключи данных, схожее с `<?php echo $name; ?>`.

В качестве альтернативы передаче полного массива данных вспомогательной функции `view` вы можете использовать метод `with` для добавления некоторых данных в шаблон. Метод `with` возвращает экземпляр объекта представления, так что вы можете продолжить связывание методов перед возвратом шаблона:

```
return view('greeting')
    ->with('name', 'Victoria')
    ->with('occupation', 'Astronaut');
```

## Общедоступные данные для всех шаблонов

Иногда требуется сделать данные общедоступными для всех шаблонов, отображаемыми вашим приложением. Вы можете сделать это, используя метод `share` фасада `View`. Как правило, вызов метода `share` осуществляется в методе `boot` поставщика служб. Вы можете добавить их в класс `App\Providers\AppServiceProvider` или создать отдельного поставщика для их размещения:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        //
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        View::share('key', 'value');
    }
}
```

## # Компоновщики шаблонов

Компоновщики шаблонов – это замыкания или методы класса, которые вызываются при отрисовке шаблонов. Если у вас есть данные, которые вы хотите привязать к шаблону каждый раз при его отрисовке, компоновщик шаблонов поможет вам

организовать эту логику в одном месте. Компоновщики шаблонов особенно полезны, если один и тот же шаблон возвращается несколькими маршрутами или контроллерами в вашем приложении и всегда требует определенного фрагмента данных.

Как правило, компоновщики шаблонов регистрируются в одном из [поставщиков служб](#) вашего приложения. В этом примере мы предположим, что в `App\Providers\AppServiceProvider` будет находиться эта логика.

Мы будем использовать метод `composer` фасада `View`, чтобы зарегистрировать компоновщик. Laravel по умолчанию не содержит каталог для классов компоновщиков, поэтому вы можете организовать их, как хотите. Например, вы можете создать каталог `app/View/Composers` для размещения всех компоновщиков вашего приложения:

```
<?php
```

```
namespace App\Providers;

use App\View\Composers\ProfileComposer;
use Illuminate\Support\Facades;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        // Использование компоновщиков на основе классов ...
        Facades\View::composer('profile', ProfileComposer::class);

        // Использование анонимных компоновщиков ...
        Facades\View::composer('welcome', function (View $view) {
            // ...
        });
    }
}
```

```
Facades\View::composer('dashboard', function (View $view) {
    // ...
});
}
}
```

Теперь, когда мы зарегистрировали компоновщик, метод `compose` класса `App\View\Composers\ProfileComposer` будет выполняться каждый раз, когда отрисовывается шаблон профиля. Давайте посмотрим на пример класса компоновщика:

```
<?php

namespace App\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * Создать нового компоновщика профиля.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Привязать данные к шаблону.
     */
    public function compose(View $view): void
    {
        $view->with('count', $this->users->count());
    }
}
```

Как видите, все компоновщики внедряются через [контейнер служб](#), поэтому вы можете указать любые зависимости, которые вам нужны, в конструкторе компоновщика.

## Связывание компоновщика с несколькими шаблонами

Вы можете связать компоновщика с несколькими шаблонами одновременно, передав массив шаблонов в качестве первого аргумента методу `composer`:

```
use App\Views\Composers\MultiComposer;
use Illuminate\Support\Facades\View;

View::composer(
    ['profile', 'dashboard'],
    MultiComposer::class
);
```

Допускается использование метасимвола подстановки `*`, что позволит вам прикрепить компоновщик ко всем шаблонам:

```
use Illuminate\Support\Facades;
use Illuminate\View\View;

Facades\View::composer('*', function (View $view) {
    // ...
});
```

## Создатели шаблонов

«Создатели» шаблонов очень похожи на компоновщиков; но, они выполняются сразу после создания экземпляра, а не ожидают отрисовки шаблона. Чтобы зарегистрировать создателя шаблона, используйте метод `creator`:

```
use App\View\Creators\ProfileCreator;
use Illuminate\Support\Facades\View;

View::creator('profile', ProfileCreator::class);
```

## # Оптимизация шаблонов

По умолчанию шаблоны Blade компилируются по требованию. Когда выполняется запрос, который отрисовывает шаблон, Laravel определит, существует ли скомпилированная версия шаблона. Если файл существует, Laravel далее определит, был ли исходный шаблон изменен позднее скомпилиированного. Если скомпилированного шаблона либо не существует, либо исходный шаблон был изменен, Laravel перекомпилирует шаблон.

Компиляция шаблонов во время запроса отрицательно влияет на производительность, поэтому Laravel содержит команду Artisan `view:cache` для предварительной компиляции всех шаблонов, используемых вашим приложением. Для повышения производительности вы можете выполнить эту команду как часть процесса развертывания:

```
php artisan view:cache
```

Вы можете использовать команду `view:clear` для очистки кеша шаблонов:

```
php artisan view:clear
```

# Шаблонизатор Blade

## # Введение

# Новый уровень Blade с помощью Livewire

## # Отображение данных

# Преобразование в HTML-сущности

# Blade и JavaScript фреймворки

## # Директивы Blade

# Операторы If

# Операторы Switch

# Циклы

# Переменная Loop

# Css-классы и стили по условию

# Дополнительные атрибуты

# Подключение дочерних шаблонов

# Директива @once

# Необработанный PHP

# Комментарии

## # Компоненты

# Отрисовка компонентов

# Индексация компонентов

# Передача данных компонентам

# Атрибуты компонента

# Зарезервированные ключевые слова

# Слоты

# Встроенные шаблоны компонентов

# Динамические компоненты

# Ручная регистрация компонентов

## # Анонимные компоненты

# Анонимные Index Компоненты

# Свойства / атрибуты данных

- # Доступ к родительским данным
- # Анонимные пути компонентов

**# Создание макетов**

- # Макеты с использованием компонентов
- # Макеты с использованием наследования шаблонов

**# Формы**

- # Поле CSRF
- # Поле Method
- # Ошибки валидации

**# Стеки**

**# Внедрение служб**

**# Рендеринг шаблонов Blade из строки**

**# Рендеринг фрагментов Blade**

**# Расширение Blade**

- # Пользовательские обработчики вывода
- # Пользовательские операторы If

## # Введение

Blade – это простой, но мощный движок шаблонов, входящий в состав Laravel. В отличие от некоторых шаблонизаторов PHP, Blade не ограничивает вас в использовании обычного “сырого” кода PHP в ваших шаблонах. На самом деле, все шаблоны Blade компилируются в обычный PHP-код и кешируются до тех пор, пока не будут изменены, что означает, что Blade добавляет фактически нулевую нагрузку вашему приложению. Файлы шаблонов Blade используют расширение файла `.blade.php` и обычно хранятся в каталоге `resources/views`.

Шаблоны Blade могут быть возвращены из маршрутов или контроллера с помощью глобального помощника `view`. Конечно, как упоминалось в документации по [HTML-шаблонам](#), данные могут быть переданы в шаблоны Blade, используя второй аргумент помощника `view`:

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

# Новый уровень Blade с помощью Livewire

Хотите повысить эффективность ваших шаблонов Blade и легко создавать динамические интерфейсы? Обратите внимание на [Laravel Livewire](#). Livewire позволяет создавать компоненты Blade, дополненные динамической функциональностью, которая обычно доступна только благодаря фреймворкам фронтенда, таким как React или Vue. Такой подход отлично подходит для создания современных, реактивных интерфейсов без сложностей, связанных с отрисовкой на клиентской стороне или сборкой, присущими многим JavaScript-фреймворкам.

## # Отображение данных

Вы можете отображать данные, которые передаются в шаблоны Blade, заключив переменную в фигурные скобки. Например, учитывая следующий маршрут:

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

Вы можете отобразить содержимое переменной `name` следующим образом:

```
Hello, {{ $name }}.
```

Выражения вывода `{{ }}` Blade автоматически отправляются через функцию `htmlspecialchars` PHP для предотвращения XSS-атак.

Вы не ограничены отображением содержимого переменных, переданных в шаблон. Вы также можете вывести результаты любой функции PHP. Фактически, вы можете поместить любой PHP-код в выражение вывода Blade:

```
The current UNIX timestamp is {{ time() }}.
```

## Преобразование в HTML-сущности

По умолчанию Blade (и Laravel функция `e`) будет дважды кодировать объекты HTML.

Если вы хотите отключить двойное кодирование, вызовите метод

`Blade::withoutDoubleEncoding` в методе `boot` вашего `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Blade::withoutDoubleEncoding();
    }
}
```

## Вывод неэкранированных данных

По умолчанию, выражения вывода `{{ }}` Blade автоматически отправляются через функцию `htmlspecialchars` PHP для предотвращения XSS-атак. Если вы не хотите, чтобы ваши данные были экранированы, вы можете использовать следующий синтаксис:

```
Hello, {!! $name !!}. 
```

Будьте очень осторожны при выводе содержимого, полученного от пользователей вашего приложения. Обычно следует использовать экранированный синтаксис двойных фигурных скобок для предотвращения атак XSS при отображении данных, предоставленных пользователем.

# Blade и JavaScript фреймворки

Поскольку во многих фреймворках JavaScript также используются «фигурные» скобки, чтобы указать, что данное выражение должно отобразиться в браузере, вы можете использовать символ @, чтобы сообщить движку Blade, что выражение должно оставаться нетронутым. Например:

```
<h1>Laravel</h1>
```

```
Hello, @{{ name }}.
```

В этом примере Blade удалит символ @; однако выражение {{ name }} останется нетронутым движком Blade, что позволит обработать его вашим фреймворком JavaScript.

Символ @ также используется для исключения из обработки директив Blade:

```
{{-- Шаблон Blade --}}
```

```
@@if()
```

```
<!-- Вывод HTML -->
```

```
@if()
```

## Вывод JSON

Иногда вы можете передать массив вашему шаблону с намерением отобразить его как JSON, чтобы инициализировать переменную JavaScript. Например:

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

Однако вместо ручного вызова `json_encode`, вы можете использовать метод `Illuminate\Support\Js::from`. Метод `from` принимает те же аргументы, что и функция PHP `json_encode`; однако это гарантирует, что полученный JSON будет правильно экранирован для включения в кавычки HTML. Метод `from` вернет строковый оператор JavaScript `JSON.parse`, который преобразует данный объект или массив в допустимый объект:

```
<script>
  var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

Последние версии приложения Laravel включают фасад `Js`, который обеспечивает удобный доступ к этой функции в ваших шаблонах Blade:

```
<script>
  var app = {{ Js::from($array) }};
</script>
```

Вы должны использовать директиву `Js::from` только для отображения существующих переменных как JSON. Шаблонизатор Blade основан на регулярных выражениях, и попытки передать сложное выражение в директиву могут вызвать неожиданные сбои.

## Директива @verbatim

Если вы отображаете переменные JavaScript в крупной части своего шаблона, вы можете заключить HTML в директиву `@verbatim`, чтобы вам не приходилось добавлять префикс `@` к каждому выражению вывода Blade:

```
@verbatim
  <div class="container">
    Hello, {{ name }}.
  </div>
@endverbatim
```

## # Директивы Blade

Помимо наследования шаблонов и отображения данных, Blade также содержит удобные псевдонимы для общих структур управления PHP, таких, как условные операторы и циклы. Эти директивы обеспечивают очень чистый и лаконичный

способ работы со структурами управления PHP, но при этом остаются схожими со своими аналогами PHP.

## Операторы If

Вы можете создавать операторы `if`, используя директивы `@if`, `@elseif`, `@else`, и `@endif`. Эти директивы работают так же, как и их аналоги в PHP:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Для удобства Blade также содержит директиву `@unless`:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

В дополнение к уже обсужденным условным директивам, директивы `@isset` и `@empty` могут использоваться в качестве удобных ярлыков для соответствующих функций PHP:

```
@isset($records)
    // Переменная $records определена и не равна `null` ...
@endisset

@empty($records)
    // Переменная $records считается «пустой» ...
@endempty
```

## Директивы аутентификации

Директивы `@auth` и `@guest` могут использоваться для быстрого определения, является ли текущий пользователь [аутентифицированным](#) или считается гостем:

```
@auth
    // Пользователь аутентифицирован ...
@endauth

@guest
    // Пользователь не аутентифицирован ...
@endguest
```

При необходимости вы можете указать охранника аутентификации для проверки при использовании директив `@auth` и `@guest`:

```
@auth('admin')
    // Пользователь аутентифицирован ...
@endauth

@guest('admin')
    // Пользователь не аутентифицирован ...
@endguest
```

## Директивы окружения

Вы можете проверить, запущено ли приложение в эксплуатационном окружении, с помощью директивы `@production`:

```
@production
    // Содержимое, отображаемое только в эксплуатационном окружении ...
@endproduction
```

Или вы можете определить, работает ли приложение в конкретной среде, с помощью директивы `@env`:

```
@env('staging')
    // Приложение запущено в «переходном» окружении ...
@endenv

@env(['staging', 'production'])
    // Приложение запущено в «переходном» или «рабочем» окружении ...
@endenv
```

## Директивы секций

Вы можете определить, есть ли в секции наследуемого шаблона содержимое, используя директиву `@hasSection`:

```
@hasSection('navigation')
<div class="pull-right">
    @yield('navigation')
</div>

<div class="clearfix"></div>
@endif
```

Вы можете использовать директиву `sectionMissing`, чтобы определить, что в секции нет содержимого:

```
@sectionMissing('navigation')
<div class="pull-right">
    @include('default-navigation')
</div>
@endif
```

## Директивы сессии

Директива `@session` может использоваться для определения существования значения сессии. Если значение сессии существует, содержимое шаблона внутри директив `@session` и `@endsession` будет оценено. Внутри содержимого директивы `@session` вы можете использовать переменную `$value` для вывода значения сессии:

```
@session('status')
<div class="p-4 bg-green-100">
    {{ $value }}
</div>
@endsession
```

## Операторы Switch

Операторы Switch могут быть созданы с использованием директив `@switch`, `@case`, `@break`, `@default` и `@endswitch`:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

## Циклы

В дополнение к условным операторам, Blade содержит простые директивы для работы со структурами циклов PHP. Опять же, каждая из этих директив работает так же, как и их аналоги в PHP:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

Во время повторения цикла 'foreach' вы можете использовать [переменную Loop](#), чтобы получить информацию о цикле, например, находитесь ли вы в первой или последней итерации цикла.

При использовании циклов вы также можете пропустить текущую итерацию или завершить цикл, используя директивы `@continue` и `@break`:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Вы также можете включить в объявление директивы условие продолжения или прерывания:

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

## Переменная Loop

Во время повторения цикла `foreach` доступна переменная `$loop`. Она обеспечивает доступ к некоторой полезной информации, например, индекс текущего цикла, первая это или последняя итерация цикла:

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif
```

```
<p>This is user {{ $user->id }}</p>
@endforeach
```

При нахождении во вложенном цикле, вы можете получить доступ к переменной `$loop` родительского цикла через свойство `parent`:

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is the first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

Переменная `$loop` также содержит множество других полезных свойств:

Свойство	Описание
<code>\$loop-&gt;index</code>	Индекс текущей итерации цикла (начинается с 0).
<code>\$loop-&gt;iteration</code>	Текущая итерация цикла (начинается с 1).
<code>\$loop-&gt;remaining</code>	Итерации, оставшиеся в цикле.
<code>\$loop-&gt;count</code>	Общее количество элементов в итерируемом массиве.
<code>\$loop-&gt;first</code>	Первая ли это итерация цикла.
<code>\$loop-&gt;last</code>	Последняя ли это итерация цикла.
<code>\$loop-&gt;even</code>	Четная ли это итерация цикла.
<code>\$loop-&gt;odd</code>	Нечетная ли это итерация цикла.
<code>\$loop-&gt;depth</code>	Уровень вложенности текущего цикла.
<code>\$loop-&gt;parent</code>	Переменная родительского цикла во вложенном цикле.

## Css-классы и стили по условию

Директива `@class` осуществляет построение строки CSS-классов исходя из заданных условий. Директива принимает массив классов, где ключ массива содержит класс или классы, которые вы хотите добавить, а значение является булевым выражением. Если элемент массива имеет числовой ключ, он всегда будет включен в отрисованный список классов:

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
    'text-gray-500' => ! $isActive,
    'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

Также, директива `@style` может использоваться, чтобы условно добавлять встроенные стили CSS к HTML-элементу:

```
@php
    $isActive = true;
@endphp

<span @style([
    'background-color: red',
    'font-weight: bold' => $isActive,
])></span>

<span style="background-color: red; font-weight: bold;"></span>
```

## Дополнительные атрибуты

Для удобства, вы можете использовать директиву `@checked`, чтобы легко указать, должен ли данный флагок HTML быть “отмеченным”. Эта директива будет выводить `checked`, если условие выполнится:

```
<input
    type="checkbox"
    name="active"
    value="active"
```

```
@checked(old('active', $user->active))  
/>>
```

Аналогично, директива `@selected` может использоваться, чтобы указать, должен ли заданный вариант выбора быть "выбранным":

```
<select name="version">  
    @foreach ($product->versions as $version)  
        <option value="{{ $version }}" @selected(old('version') == $version)>  
            {{ $version }}  
        </option>  
    @endforeach  
</select>
```

Кроме того, директива `@disabled` может использоваться, чтобы указать, должен ли данный элемент быть "отключенным":

```
<button type="submit" @disabled($errors->isNotEmpty())>Отправить</button>
```

Более того, директива `@readonly` может быть использована, чтобы указать, должен ли данный элемент быть "только для чтения":

```
<input  
    type="email"  
    name="email"  
    value="email@laravel.com"  
    @readonly($user->isNotAdmin())  
/>
```

Кроме того, директива `@required` может быть использована, чтобы указать, должен ли данный элемент быть "обязательным":

```
<input  
    type="text"  
    name="title"  
    value="title"  
    @required($user->isAdmin())  
/>
```

# Подключение дочерних шаблонов

Хотя вы можете использовать директиву `@include`, [компоненты](#) Blade содержат аналогичный функционал и предлагают несколько преимуществ по сравнению с директивой `@include`, например привязку данных и атрибутов.

Директива `@include` Blade позволяет вам включать шаблоны из другого шаблона. Все переменные, доступные для родительского шаблона, будут доступны для включенного шаблона:

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Включенный шаблон унаследует все данные, доступные в родительском шаблоне, но вы также можете передать массив дополнительных данных, которые должны быть доступны для включенного шаблона:

```
@include('view.name', ['status' => 'complete'])
```

Если вы попытаетесь включить несуществующий шаблон, Laravel выдаст ошибку. Если вы хотите включить шаблон, который может присутствовать или отсутствовать, вам следует использовать директиву `@includeIf`:

```
@includeIf('view.name', ['status' => 'complete'])
```

Если вы хотите включить шаблон в зависимости от результата логического выражения, возвращающего либо `true`, либо `false`, то используйте директивы `@includeWhen` и `@includeUnless`:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])
```

```
@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

Чтобы включить первый существующий шаблон из переданного массива шаблонов, вы можете использовать директиву `includeFirst`:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

Вам следует избегать использования в ваших шаблонах Blade констант `__DIR__` и `__FILE__`, поскольку они будут ссылаться на расположение кешированного, скомпилированного шаблона.

## Отрисовка шаблонов с коллекциями

Вы можете скомбинировать циклы и подключение шаблона в одну строку с помощью директивы Blade `@each`:

```
@each('view.name', $jobs, 'job')
```

Первый аргумент директивы `@each` – это шаблон, отображаемый для каждого элемента в массиве или коллекции. Второй аргумент – это массив или коллекция, которую вы хотите перебрать. Третий аргумент – это имя переменной, которая будет присвоена текущей итерации в шаблоне. Так, например, если вы выполняете итерацию по массиву `jobs`, обычно вам нужно обращаться к каждому элементу как к переменной `job` в шаблоне. Ключ массива для текущей итерации будет доступен как переменная `key` в шаблоне.

Вы можете передать четвертый аргумент директиве `@each`. Этот аргумент определяет шаблон, который будет отображаться, если переданный массив пуст.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

Шаблоны, отображаемые с помощью `@each`, не наследуют переменные родительского шаблона. Если дочернему шаблону требуются эти переменные, вам следует использовать вместо них директивы `@foreach` и `@include`.

## Директива `@once`

Директива `@once` позволяет вам определить часть шаблона, которая будет проанализирована только один раз за цикл визуализации. Это может быть полезно для вставки переданного фрагмента JavaScript в подвал страницы с помощью [стеков](#). Например, если вы отображаете переданный [компонент](#) в цикле, то бывает необходимо разместить JavaScript в подвале при визуализации компонента только единожды:

```
@once
  @push('scripts')
    <script>
      // Ваш JavaScript...
    </script>
  @endpush
@endonce
```

Поскольку директива `@once` часто используется в сочетании с директивами `@push` или `@prepend`, для удобства доступны директивы `@pushOnce` и `@prependOnce`:

```
@pushOnce('scripts')
  <script>
    // Ваш JavaScript...
  </script>
@endPushOnce
```

## Необработанный PHP

В крайних ситуациях можно встроить PHP-код в ваши шаблоны. Вы можете использовать директиву `@php` Blade для размещения блока простого PHP в вашем шаблоне:

```
@php  
    $counter = 1;  
@endphp
```

Или, если вам нужно использовать только PHP для импорта класса, вы можете использовать директиву `@use`:

```
@use('App\Models\Flight')
```

Второй аргумент может быть использован в директиве `@use` для указания псевдонима импортируемого класса:

```
@use('App\Models\Flight', 'FlightModel')
```

## Комментарии

Blade также позволяет вам определять комментарии в ваших шаблонах. Однако, в отличие от комментариев HTML, комментарии Blade не будут включены в результирующий HTML, возвращаемый вашим приложением:

```
{{-- This comment will not be present in the rendered HTML --}}
```

## # Компоненты

Компоненты и слоты предоставляют те же преимущества, что и секции, макеты и включение шаблона из другого шаблона; однако, некоторым может быть легче понять мысленную модель компонентов и слотов. Есть два подхода к написанию компонентов: компоненты на основе классов и анонимные компоненты.

Чтобы создать компонент на основе класса, вы можете использовать команду `make:component Artisan`. Чтобы проиллюстрировать, как использовать компоненты, мы создадим простой компонент `Alert`. Команда `make:component` поместит компонент в каталог `app/View/Components`:

```
php artisan make:component Alert
```

Команда `make: component` также создаст шаблон для компонента. Шаблон будет помещен в каталог `resources/views/components`. При написании компонентов для вашего собственного приложения компоненты автоматически обнаруживаются в каталогах `app/View/Components` и `resources/views/components`, поэтому дополнительная регистрация компонентов обычно не требуется.

Вы также можете создавать компоненты в подкаталогах:

```
php artisan make:component Forms/Input
```

Приведенная выше команда создаст компонент `Input` в каталоге `app/View/Components/Forms`, а шаблон будет помещен в каталог `resources/views/components/forms`.

Если вы хотите создать анонимный компонент (компонент только с шаблоном в Blade без класса), вы можете использовать флаг `--view` при вызове команды `make:component`:

```
php artisan make:component forms.input --view
```

Вышеприведенная команда создаст файл Blade по пути `resources/views/components/forms/input.blade.php`, который может быть отображен как компонент с помощью `<x-forms.input />`.

## Самостоятельная регистрация компонентов пакета

При написании компонентов для вашего собственного приложения компоненты автоматически обнаруживаются в каталогах `app/View/Components` и `resources/views/components`.

Однако, если вы создаете пакет, который использует компоненты Blade, вам необходимо вручную зарегистрировать класс компонента и его псевдоним HTML-тега. Вы должны зарегистрировать свои компоненты в методе `boot` поставщика служб вашего пакета:

```
use Illuminate\Support\Facades\Blade;  
  
/**
```

```
* Загрузка любых служб пакета.  
*/  
public function boot(): void  
{  
    Blade::component('package-alert', Alert::class);  
}
```

После того как ваш компонент был зарегистрирован, он может быть отображен с использованием псевдонима тега:

```
<x-package-alert/>
```

Как вариант, вы можете использовать метод `componentNamespace` для автоматической загрузки классов компонентов по соглашению. Например, пакет `Nightshade` может иметь компоненты `Calendar` и `ColorPicker`, которые находятся в пространстве имен `Package\Views\Components`:

```
use Illuminate\Support\Facades\Blade;  
  
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void  
{  
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');  
}
```

Это позволит использовать компоненты пакета в пространстве имен их поставщиков, используя синтаксис `x-package-name::`:

```
<x-nightshade::calendar />  
<x-nightshade::color-picker />
```

Blade автоматически обнаружит класс, связанный с этим компонентом, используя «верблюжий регистр» имени компонента. Подкаталоги также поддерживаются с использованием «точечной» нотации.

## Отрисовка компонентов

Для отображения компонента вы можете использовать тег компонента Blade в одном из ваших шаблонов Blade. Теги компонентов Blade начинаются со строки `x-`, за которой следует имя в «шашлычном регистре» класса компонента:

```
<x-alert/>  
<x-user-profile/>
```

Если класс компонента имеет вложенность в каталоге `app/View/Components`, то вы можете использовать символ `.` для обозначения вложенности каталогов.

Например, если мы предполагаем, что компонент находится в `app/View/Components/Inputs/Button.php`, то мы можем отобразить его так:

```
<x-inputs.button/>
```

Если вы хотите выборочно отображать ваш компонент, вы можете указать метод `shouldRender` в классе вашего компонента. Если результат метода `shouldRender` равен `false`, то компонент не будет отображаться:

```
use Illuminate\Support\Str;  
  
/**  
 * Определяет, должен ли компонент отображаться  
 */  
public function shouldRender(): bool  
{  
    return Str::length($this->message) > 0;  
}
```

## Индексация компонентов

Иногда компоненты являются частью группы компонентов, и вам может потребоваться сгруппировать связанные компоненты в одном каталоге. Например, представьте себе компонент «card» со следующей структурой классов:

```
App\Views\Components\Card\Card  
App\Views\Components\Card\Header  
App\Views\Components\Card\Body
```

Поскольку корневой компонент `Card` вложен в каталог `Card`, вы можете ожидать, что вам потребуется визуализировать компонент через `<x-card.card>`. Однако, когда имя файла компонента совпадает с именем каталога компонента, Laravel автоматически предполагает, что компонент является «корневым» компонентом, и позволяет отображать компонент без повторения имени каталога:

```
<x-card>
  <x-card.header>...</x-card.header>
  <x-card.body>...</x-card.body>
</x-card>
```

## Передача данных компонентам

Вы можете передавать данные в компоненты Blade, используя атрибуты HTML. Жестко запрограммированные примитивные значения могут быть переданы компоненту с помощью простых строк атрибутов HTML. Выражения и переменные PHP следует передавать компоненту через атрибуты, которые используют символ `:` в качестве префикса:

```
<x-alert type="error" :message="$message"/>
```

Вы должны определить необходимые данные компонента в его конструкторе класса. Все общедоступные свойства компонента будут автоматически доступны в шаблоне компонента. Нет необходимости передавать данные в шаблон из метода `render` компонента:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;
use Illuminate\View\View;

class Alert extends Component
{
    public function __construct(
        public string $type,
        public string $message,
    ) {}

    /**
     * @param string $type
     * @param string $message
     */
    public function render(): View
    {
        return view('components.alert');
    }
}
```

```
* Получить шаблон / содержимое, представляющее компонент.  
*/  
public function render(): View  
{  
    return view('components.alert');  
}  
}
```

Когда ваш компонент визуализируется, вы можете отображать содержимое общедоступных переменных вашего компонента, выводя переменные по имени:

```
<div class="alert alert-{{ $type }}>  
    {{ $message }}  
</div>
```

## Именование

Аргументы конструктора компонентов следует указывать с помощью [camelCase](#), а при обращении к именам аргументов в ваших атрибутах HTML следует использовать [kebab-case](#). Например, учитывая следующий конструктор компонента:

```
/**  
 * Создать экземпляр компонента.  
 */  
public function __construct(  
    public string $alertType,  
) {}
```

Аргумент `$alertType` может быть передан компоненту следующим образом:

```
<x-alert alert-type="danger" />
```

## Сокращенный синтаксис атрибутов

При передаче атрибутов компонентам вы также можете использовать “сокращенный синтаксис атрибутов”. Это удобно, поскольку имена атрибутов часто совпадают с именами переменных, к которым они относятся:

```
{-- Сокращенный синтаксис атрибутов... --}
<x-profile :$userId :$name />
{-- Эквивалентно... --}
<x-profile :user-id="$userId" :name="$name" />
```

## Экранирование атрибутов от синтаксического анализа

Поскольку некоторые фреймворки JavaScript, такие, как Alpine.js, также используют атрибуты с префиксом двоеточия, вы можете использовать префикс с двойным двоеточием (::), чтобы сообщить Blade, что атрибут не является выражением PHP. Например, учитывая следующий компонент:

```
<x-button ::class="{ danger: isDeleting }">
    Submit
</x-button>
```

Blade отобразит следующий HTML-код:

```
<button :class="{ danger: isDeleting }">
    Submit
</button>
```

## Методы компонента

В дополнение к общедоступным переменным, доступным для вашего шаблона компонента, могут быть вызваны любые общедоступные методы компонента. Например, представьте компонент, у которого есть метод `isSelected`:

```
/**
 * Определить, является ли переданная опция выбранной.
 */
public function isSelected(string $option): bool
{
    return $option === $this->selected;
}
```

Вы можете выполнить этот метод из своего шаблона компонента, вызвав переменную, соответствующую имени метода:

```
<option {{ $isSelected($value) ? 'selected="selected"' : '' }} value="{{ $value }}>
{{ $label }}
</option>
```

## Доступ к атрибутам и слотам в классах компонентов

Компоненты Blade также позволяют получить доступ к имени компонента, атрибутам и слоту внутри метода `render` класса. Однако, чтобы получить доступ к этим данным, вы должны вернуть замыкание из метода `render` вашего компонента:

```
use Closure;

/**
 * Получить шаблон / содержимое, представляющее компонент.
 */
public function render(): Closure
{
    return function () {
        return '<div {{ $attributes }}>Components content</div>';
    };
}
```

Замыкание, возвращаемое методом `render` вашего компонента, также может получать массив `$data` в качестве единственного аргумента. Этот массив будет содержать несколько элементов, предоставляющих информацию о компоненте:

```
return function (array $data) {
    // $data['componentName'];
    // $data['attributes'];
    // $data['slot'];

    return '<div {{ $attributes }}>Components content</div>';
}
```

Элементы массива `$data` никогда не должны быть непосредственно встроены в строку Blade, возвращающую вашим методом `render`, так как это

может привести к удаленному выполнению кода через вредоносное содержимое атрибута.

`componentName` эквивалентно имени, используемому в HTML-теге после префикса `x-`. Таким образом, `componentName` компонента `<x-alert />` будет `alert`. Элемент `attributes` будет содержать все атрибуты, которые присутствовали в HTML-теге. Элемент `slot` – это экземпляр `Illuminate\Support\HtmlString` с содержимым слота компонента.

Замыкание должно возвращать строку. Если возвращенная строка соответствует существующему шаблону, то этот шаблон будет отрисован; в противном случае возвращенная строка будет оцениваться как встроенный шаблон Blade.

## Дополнительные зависимости

Если вашему компоненту требуются зависимости из [контейнера служб](#) Laravel, то вы можете указать их перед любыми атрибутами данных компонента, и они будут автоматически внедрены контейнером:

```
use App\Services\AlertCreator

/**
 * Создать экземпляр компонента.
 */
public function __construct(
    public AlertCreator $creator,
    public string $type,
    public string $message,
) {}
```

## Скрытие атрибутов / методов

Если вы хотите, чтобы некоторые публичные методы или свойства не использовались как переменные в шаблоне компонента, вы можете добавить их в свойство массива `$except` в вашем компоненте:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;
```

```

class Alert extends Component
{
    /**
     * Свойства / методы, которые не должны использоваться в шаблоне компонента.
     *
     * @var array
     */
    protected $except = ['type'];

    /**
     * Create the component instance.
     */
    public function __construct(
        public string $type,
    ) {}
}

```

## Атрибуты компонента

Мы уже рассмотрели, как передавать атрибуты данных в компонент; иногда требуется указать дополнительные атрибуты HTML, такие, как `class`, которые не являются частью данных, необходимых для функционирования компонента. Как правило, вы хотите передать эти дополнительные атрибуты корневому элементу шаблона компонента. Например, представьте, что мы хотим отобразить компонент `alert` следующим образом:

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

Все атрибуты, которые не являются частью конструктора компонента, будут автоматически добавлены в «коллекцию атрибутов» компонента. Эта коллекция атрибутов автоматически становится доступной для компонента через переменную `$attributes`. Все атрибуты могут отображаться в компоненте путем вывода этой переменной:

```

<div {{ $attributes }}>
    <!-- Component content -->
</div>

```

Использование таких директив, как `@env` в тегах компонентов в настоящее время не поддерживается. Например, `<x-alert :live="@env('production')"/>` не будет компилироваться.

## Атрибуты по умолчанию и слияние атрибутов

Иногда требуется указать значения по умолчанию для атрибутов или добавить дополнительные значения в некоторые атрибуты компонента. Для этого вы можете использовать метод `merge` коллекции атрибутов. Этот метод особенно полезен для определения набора CSS-классов по умолчанию, которые всегда должны применяться к компоненту:

```
<div {{ $attributes->merge(['class' => 'alert alert-' . $type]) }}>
    {{ $message }}
</div>
```

Если предположить,

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Окончательный обработанный HTML-код компонента будет выглядеть следующим образом:

```
<div class="alert alert-error mb-4">
    <!-- Contents of the $message variable --&gt;
&lt;/div&gt;</pre>
```

## Условное слияние классов

По желанию можно объединить классы, если указанное условие истинно. Вы можете сделать это с помощью метода `class`, принимающий массив классов, где ключ массива содержит класс или классы, которые вы хотите добавить, а значение

является логическим выражением. Если элемент массива имеет числовой ключ, то он всегда будет добавлен в конечный список классов:

```
<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>  
    {{ $message }}  
</div>
```

Если вам нужно объединить другие атрибуты в свой компонент, вы можете связать метод `merge` с методом `class`:

```
<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>  
    {{ $slot }}  
</button>
```

Если вам нужно условно скомпилировать классы для других элементов HTML, которые не должны получать объединенные атрибуты, вы можете использовать директиву `@class`.

## Слияние неклассовых атрибутов

При слиянии атрибутов, которые не являются атрибутами класса, значения, предоставленные методу `merge`, будут считаться значениями атрибута по умолчанию. Однако, в отличие от атрибута `class`, эти атрибуты не будут объединены с указанными значениями атрибутов. Вместо этого они будут перезаписаны. Например, реализация компонента `button` может выглядеть следующим образом:

```
<button {{ $attributes->merge(['type' => 'button']) }}>  
    {{ $slot }}  
</button>
```

Чтобы отобразить компонент кнопки с настраиваемым `type`, его можно указать при использовании компонента. Если тип не указан, будет использоваться тип `button`, определенный по умолчанию:

```
<x-button type="submit">  
    Submit  
</x-button>
```

Обработанный HTML-код компонента `button` в этом примере будет:

```
<button type="submit">  
    Submit  
</button>
```

Если вы хотите, чтобы атрибут, отличный от `class`, имел значение по умолчанию и указанное значение, объединенные вместе, вы можете использовать метод `prepends`. В этом примере атрибут `data-controller` всегда будет начинаться с `profile-controller`, а любые дополнительные указанные значения `data-controller` будут помещены после этого значения по умолчанию:

```
<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-conti  
    {{ $slot }}  
</div>
```



## Получение и фильтрация атрибутов

Вы можете фильтровать атрибуты, используя метод `filter`. Этот метод принимает замыкание, которое должно возвращать `true`, если вы хотите сохранить атрибут в коллекции атрибутов:

```
{{ $attributes->filter(fn ($value, $key) => $key == 'foo') }}
```

Для удобства вы можете использовать метод `whereStartsWith` для получения всех атрибутов, ключи которых начинаются с указанной строки:

```
{{ $attributes->whereStartsWith('wire:model') }}
```

И наоборот, метод `whereDoesntStartWith` может быть использован для исключения всех атрибутов, ключи которых начинаются с указанной строки:

```
 {{ $attributes->whereDoesntStartWith('wire:model') }}
```

Используя метод `first`, вы можете отобразить первый атрибут в указанной коллекции атрибутов:

```
 {{ $attributes->whereStartsWith('wire:model')->first() }}
```

Если вы хотите проверить, присутствует ли атрибут в компоненте, вы можете использовать метод `has`. Этот метод принимает имя атрибута в качестве единственного аргумента и возвращает логическое значение, указывающее, присутствует ли атрибут:

```
@if ($attributes->has('class'))  
    <div>Атрибут class присутствует</div>  
@endif
```

Если передан массив в метод `has`, то он проверит, присутствуют ли все указанные атрибуты у компонента:

```
@if ($attributes->has(['name', 'class']))  
    <div>Все указанные атрибуты присутствуют</div>  
@endif
```

Метод `hasAny` может быть использован для определения, присутствует ли хотя бы один из указанных атрибутов у компонента:

```
@if ($attributes->hasAny(['href', ':href', 'v-bind:href']))  
    <div>Один из указанных атрибутов присутствует</div>  
@endif
```

Вы можете получить значение конкретного атрибута, используя метод `get`:

```
 {{ $attributes->get('class') }}
```

## Зарезервированные ключевые слова

По умолчанию некоторые ключевые слова зарезервированы для внутреннего использования Blade при визуализации компонентов. Следующие ключевые слова не могут быть определены как публичные свойства или имена методов в ваших компонентах:

- `data`
- `render`
- `resolveView`
- `shouldRender`
- `view`
- `withAttributes`
- `withName`

## Слоты

Вам часто потребуется передавать дополнительный контент вашему компоненту через «слоты». Слоты компонентов отображаются путем вывода переменной `$slot`. Чтобы изучить эту концепцию, представим, что компонент `alert` имеет следующую разметку:

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Мы можем передавать контент в `slot`, вставив контент в компонент:

```
<x-alert>
    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Иногда компоненту может потребоваться отрисовать несколько разных слотов в разных местах внутри компонента. Давайте модифицируем наш компонент оповещения, чтобы учесть вставку слота `title`:

```
<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Вы можете определить содержимое именованного слота с помощью тега `x-slot`. Любой контент, не указанный в явном теге `x-slot`, будет передан компоненту в переменной `$slot`:

```
<x-alert>
    <x-slot:title>
        Server Error
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Вы можете вызвать метод слота `isEmpty`, чтобы определить, содержит ли он контент:

```
<span class="alert-title">{{ $title }}</span>
<div class="alert alert-danger">
    @if ($slot->isEmpty())
        This is default content if the slot is empty.
    @else
        {{ $slot }}
    @endif
</div>
```

Кроме того, метод `hasActualContent` может быть использован для определения, содержит ли слот какой-либо «фактический» контент, не являющийся HTML-комментарием:

```
@if ($slot->hasActualContent())
    The scope has non-comment content.
@endif
```

## Слоты с ограниченной областью видимости

Если вы использовали фреймворк JavaScript, такой, как Vue, то вы, возможно, знакомы со «слотами с ограниченной областью видимости», которые позволяют получать доступ к данным или методам из компонента в вашем слоте. Вы можете добиться аналогичного поведения в Laravel, определив общедоступные методы или свойства в вашем компоненте и получив доступ к компоненту в вашем слоте через переменную `$component`. В этом примере мы предположим, что компонент `x-alert` имеет общедоступный метод `formatAlert`, определенный в его классе компонента:

```
<x-alert>
  <x-slot:title>
    {{ $component->formatAlert('Server Error') }}
  </x-slot>

  <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

## Атрибуты слотов

Как и в компонентах Blade, вы можете назначать слотам дополнительные [атрибуты](#), например, имена классов CSS:

```
<x-card class="shadow-sm">
  <x-slot:heading class="font-bold">
    Heading
  </x-slot>

  Content

  <x-slot:footer class="text-sm">
    Footer
  </x-slot>
</x-card>
```

Чтобы взаимодействовать с атрибутами слота, можно обратиться к свойству `attributes` переменной слота. Для получения дополнительной информации о том, как взаимодействовать с атрибутами, обратитесь к документации по [атрибутам компонентов](#):

```

@props([
    'heading',
    'footer',
])

<div {{ $attributes->class(['border']) }}>
    <h1 {{ $heading->attributes->class(['text-lg']) }}>
        {{ $heading }}
    </h1>

    {{ $slot }}

    <footer {{ $footer->attributes->class(['text-gray-700']) }}>
        {{ $footer }}
    </footer>
</div>

```

## Встроенные шаблоны компонентов

Для очень маленьких компонентов может показаться обременительным управлять как классом компонента, так и шаблоном компонента. По этой причине вы можете вернуть разметку компонента прямо из метода `render`:

```

/**
 * Получить шаблон / содержимое, представляющее компонент.
 */
public function render(): string
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    blade;
}

```

## Генерация компонентов со встроенными шаблонами

Чтобы создать компонент, который использует встроенный шаблон, вы можете использовать параметр `--inline` при выполнении команды `make:component`:

```
php artisan make:component Alert --inline
```

## Динамические компоненты

Иногда вам может понадобиться отрисовать компонент, но вы не знаете, какой компонент должен быть отрисован на этапе выполнения. В этой ситуации вы можете использовать встроенный компонент `dynamic-component` в Laravel, чтобы отрисовать компонент на основе значения или переменной, полученных на этапе выполнения:

```
// $componentName = "secondary-button";
<x-dynamic-component :component="$componentName" class="mt-4" />
```

## Ручная регистрация компонентов

Следующая документация по ручной регистрации компонентов в основном применима к тем, кто пишет пакеты Laravel, включающие компоненты отображения. Если вы не пишете пакет, эта часть документации по компонентам может быть неактуальной для вас.

Когда вы создаете компоненты для своего приложения, они автоматически обнаруживаются в каталоге `app/View/Components` и каталоге `resources/views/components`.

Однако, если вы создаете пакет, использующий компоненты Blade, или помещаете компоненты в нестандартные каталоги, вам потребуется вручную зарегистрировать класс вашего компонента и его псевдоним HTML-тега, чтобы Laravel знал, где найти компонент. Обычно вы регистрируете свои компоненты в методе `boot` сервис-провайдера вашего пакета:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * Инициализация сервисов вашего пакета.
 */
public function boot(): void
```

```
{  
    Blade::component('package-alert', AlertComponent::class);  
}
```

После того, как ваш компонент будет зарегистрирован, его можно отрисовать с использованием его псевдонима тега:

```
<x-package-alert/>
```

## Автозагрузка компонентов пакета

В качестве альтернативы вы можете использовать метод `componentNamespace` для автозагрузки классов компонентов согласно конвенции. Например, пакет `Nightshade` может содержать компоненты `Calendar` и `ColorPicker`, которые находятся в пространстве имен `Package\Views\Components`:

```
use Illuminate\Support\Facades\Blade;  
  
/**  
 * Инициализация сервисов вашего пакета.  
 */  
public function boot(): void  
{  
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');  
}
```

Это позволит использовать компоненты пакета с использованием пространства имен вендора с использованием синтаксиса `package-name::`:

```
<x-nightshade::calendar />  
<x-nightshade::color-picker />
```

Blade автоматически обнаружит класс, связанный с этим компонентом, преобразовав имя компонента в паскальный регистр. Также поддерживаются подкаталоги, используя синтаксис “точка”.

## # Анонимные компоненты

Подобно встроенным компонентам, анонимные компоненты предоставляют механизм для управления компонентом через один файл. Однако анонимные компоненты используют один файл шаблона, но не имеют связанного с компонентом класса. Чтобы определить анонимный компонент, вам нужно только разместить шаблон Blade в вашем каталоге `resources/views/components`. Например, если вы определили компонент в `resources/views/components/alert.blade.php`, вы можете просто отобразить его так:

```
<x-alert/>
```

Вы можете использовать символ `.`, чтобы указать, вложен ли компонент в каталоге `components`. Например, если компонент определен в `resources/views/components/inputs/button.blade.php`, вы можете отобразить его так:

```
<x-inputs.button/>
```

## Анонимные Index Компоненты

Иногда, когда компонент состоит из множества шаблонов Blade, вы можете захотеть сгруппировать шаблоны данного компонента в одном каталоге. Например, представьте компонент “аккордеон” со следующей структурой каталогов:

```
/resources/views/components/accordion.blade.php  
/resources/views/components/accordion/item.blade.php
```

Такая структура каталога позволяет отобразить компонент аккордеона и его элемент следующим образом:

```
<x-accordion>  
  <x-accordion.item>  
    ...  
  </x-accordion.item>  
</x-accordion>
```

Однако, чтобы отобразить компонент аккордеона через `x-accordion`, мы были вынуждены поместить шаблон компонента аккордеона “index” в каталог

`resources/views/components`, вместо того, чтобы вложить его в каталог `accordion` с другими шаблонами, связанными с аккордеоном.

К счастью, Blade позволяет вам разместить файл, соответствующий имени каталога компонента, внутри самого каталога компонента. Если этот шаблон существует, его можно отобразить как «корневой» элемент компонента, даже если он вложен в каталог. Итак, мы можем продолжать использовать тот же синтаксис Blade, что и в примере выше; однако мы изменим структуру наших каталогов следующим образом:

```
/resources/views/components/accordion/accordion.blade.php  
/resources/views/components/accordion/item.blade.php
```

## Свойства / атрибуты данных

Поскольку анонимные компоненты не имеют ассоциированного класса, вы можете задаться вопросом, как можно различить, какие данные должны быть переданы компоненту как переменные, а какие атрибуты должны быть помещены в [коллекцию атрибутов](#) компонента.

Вы можете указать, какие атрибуты следует рассматривать как переменные данных, используя директиву `@props` в верхней части шаблона Blade вашего компонента. Все остальные атрибуты компонента будут доступны через коллекцию атрибутов компонента. Если вы хотите присвоить переменной данных значение по умолчанию, вы можете указать имя переменной в качестве ключа массива и значение по умолчанию в качестве значения массива:

```
<!-- /resources/views/components/alert.blade.php -->  
  
@props(['type' => 'info', 'message'])  
  
<div {{ $attributes->merge(['class' => 'alert alert-' . $type]) }}>  
    {{ $message }}  
</div>
```

Учитывая приведенное выше определение компонента, мы можем отобразить компонент следующим образом:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

# Доступ к родительским данным

Иногда вам может потребоваться доступ к данным из родительского компонента внутри дочернего компонента. В этих случаях вы можете использовать директиву `@aware`. Например, представьте, что мы создаем сложный компонент меню, состоящий из родительского `<x-menu>` и дочернего `<x-menu.item>`:

```
<x-menu color="purple">
  <x-menu.item>...</x-menu.item>
  <x-menu.item>...</x-menu.item>
</x-menu>
```

Компонент `<x-menu>` может иметь следующую реализацию:

```
<!-- /resources/views/components/menu/index.blade.php -->

@props(['color' => 'gray'])

<ul {{ $attributes->merge(['class' => 'bg-'.$color.'-200']) }}>
  {{ $slot }}
</ul>
```

Поскольку свойство `color` было передано только родительскому элементу (`<x-menu>`), оно не будет доступно внутри `<x-menu.item>`. Однако, если мы воспользуемся директивой `@aware`, мы можем сделать ее доступной и внутри `<x-menu.item>`:

```
<!-- /resources/views/components/menu/item.blade.php -->

@aware(['color' => 'gray'])

<li {{ $attributes->merge(['class' => 'text-'.$color.'-800']) }}>
  {{ $slot }}
</li>
```

Директива `@aware` не может обращаться к родительским данным, которые не явно передаются в родительский компонент через HTML-атрибуты. Значения `@props`, которые не явно передаются в

родительский компонент, не могут быть доступны директиве `@aware`.

## Анонимные пути компонентов

Как уже обсуждалось ранее, для определения анонимных компонентов обычно использовался шаблон Blade, размещаемый в директории `resources/views/components` вашего Laravel-приложения. Однако, возможно, вам потребуется указать другие пути для анонимных компонентов, дополнительно к пути по умолчанию.

Метод `anonymousComponentPath` принимает первым аргументом “путь” к расположению анонимного компонента, а вторым аргументом – необязательное “пространство имён” для компонентов. Чаще всего этот метод вызывается из метода `boot` одного из ваших [провайдеров служб](#):

```
/**
 * Инициализация сервисов приложения.
 */
public function boot(): void
{
    Blade::anonymousComponentPath(__DIR__.'/../components');
}
```

Если пути компонентов зарегистрированы без указания префикса, как в приведенном выше примере, то компоненты можно использовать в вашем коде Blade без указания соответствующего префикса. Например, если компонент `panel.blade.php` существует в указанном пути, его можно использовать следующим образом:

```
<x-panel />
```

Вы также можете предоставить “пространство имён” вторым аргументом метода `anonymousComponentPath`:

```
Blade::anonymousComponentPath(__DIR__.'/../components', 'dashboard');
```

Когда используется префикс, компоненты из этого “пространства имён” можно использовать с префиксом пространства имён и имени компонента:

```
<x-dashboard::panel />
```

## # Создание макетов

### Макеты с использованием компонентов

Большинство веб-приложений поддерживают одинаковый общий макет на разных страницах. Было бы невероятно громоздко и сложно поддерживать наше приложение, если бы нам приходилось повторять весь HTML-макет в каждом создаваемом экране. К счастью, этот макет удобно определить как один [компонент Blade](#), а затем использовать его во всем приложении.

### Определение компонента макета

Например, представьте, что мы создаем приложение со списком задач. Мы могли бы определить компонент `layout`, который выглядит следующим образом:

```
<!-- resources/views/components/layout.blade.php -->

<html>
<head>
    <title>{{ $title ?? 'Todo Manager' }}</title>
</head>
<body>
    <h1>Todos</h1>
    <hr/>
    {{ $slot }}
</body>
</html>
```

### Использование компонента макета

Как только компонент `layout` определен, мы можем создать шаблон Blade, который будет использовать этот компонент. В этом примере мы определим простой шаблон, который отображает наш список задач:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    @foreach ($tasks as $task)
        <div>{{ $task }}</div>
    @endforeach
</x-layout>
```

Помните, что содержимое, внедренное в компонент, по умолчанию будет передано переменной `$slot` компонента `layout`. Как вы могли заметить, наш `layout` также учитывает слот `$title`, если он предусмотрен; в противном случае отображается заголовок по умолчанию. Мы можем добавить другой заголовок из нашего шаблона списка задач, используя стандартный синтаксис слотов, описанный в [документации по компонентам](#):

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    <x-slot name="title">
        Custom Title
    </x-slot>

    @foreach ($tasks as $task)
        <div>{{ $task }}</div>
    @endforeach
</x-layout>
```

Теперь, когда мы определили наш макет и шаблоны списка задач, нам просто нужно вернуть представление `task` из маршрута:

```
use App\Models\Task;
```

```
Route::get('/tasks', function () { return view('tasks', ['tasks' => Task::all()]); });
```

## Макеты с использованием наследования шаблонов

### Определение макета

Макеты также могут быть созданы с помощью «наследования шаблонов». Это был основной способ создания приложений до появления [компонентов](#).

Для начала рассмотрим простой пример. Сначала мы изучим макет страницы. Поскольку большинство веб-приложений поддерживают одинаковый общий макет на разных страницах, удобно определить этот макет как единый шаблон Blade:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
<head>
    <title>App Name - @yield('title')</title>
</head>
<body>
    @section('sidebar')
        This is the master sidebar.
    @show

    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

Как видите, этот файл содержит типичную разметку HTML. Однако, обратите внимание на директивы `@section` и `@yield`. Директива `@section`, как следует из названия, определяет секцию содержимого, тогда как директива `@yield` используется для отображения содержимого секции, предоставленного дочерним шаблоном.

Теперь, когда мы определили макет для нашего приложения, давайте определим дочернюю страницу, которая наследует макет.

## Расширение макета

При определении дочернего шаблона используйте директиву Blade `@extends`, чтобы указать, какой макет дочерний шаблон должен «наследовать». Шаблоны, расширяющие макет Blade, могут добавлять содержимое в секции макета с помощью директив `@section`. Помните, как видно из приведенного выше примера, содержимое этих секций будет отображаться в макете с помощью `@yield`:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')
```

```
@section('title', 'Page Title')

@section('sidebar')
@parent

<p>This is appended to the master sidebar.</p>
@endsection

@section('content')
<p>This is my body content.</p>
@endsection
```

В этом примере секция `sidebar` использует директиву `@parent` для добавления (а не перезаписи) содержимого к боковой панели макета. Директива `@parent` будет заменена содержимым макета при визуализации представления.

В отличие от предыдущего примера, нынешняя секция `sidebar` заканчивается `@endsection` вместо `@show`. Директива `@endsection` будет только определять секцию, в то время как `@show` будет определять и **немедленно дополнять** секцию.

Директива `@yield` также принимает значение по умолчанию в качестве второго параметра. Это значение будет отображено, если дополняемый раздел не определен:

```
@yield('content', 'Default content')
```

## # Формы

### Поле CSRF

Каждый раз, когда вы определяете HTML-форму в своем приложении, вы должны включать в форму скрытое поле токена CSRF, чтобы посредник [защиты от CSRF](#) мог провалидировать запрос. Вы можете использовать директиву `@csrf` Blade для генерации поля токена:

```
<form method="POST" action="/profile">
@csrf

...
</form>
```

## Поле Method

Поскольку HTML-формы не могут выполнять запросы `PUT`, `PATCH` или `DELETE`, вам нужно будет добавить скрытое поле `_method`, чтобы подменить эти HTTP-методы. Директива `@method` Blade поможет создать для вас такое поле:

```
<form action="/foo/bar" method="POST">
@method('PUT')

...
</form>
```

## Ошибки валидации

Директива `@error` используется для быстрой проверки наличия [сообщений об ошибках валидации](#) для конкретного атрибута. В директиве `@error` вы можете вывести содержимое переменной `$message` для отображения сообщения об ошибке:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input
    id="title"
    type="text"
    class="@error('title') is-invalid @enderror"
/>

@error('title')
<div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Поскольку директива `@error` компилируется в оператор “if”, вы можете использовать директиву `@else` для вывода содержимого, когда для атрибута

нет ошибки:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input
    id="email"
    type="email"
    class="@error('email') is-invalid @else is-valid @enderror"
/>
```

Вы можете передать **имя конкретной коллекции ошибок** в качестве второго параметра директивы `@error` для получения сообщений об ошибках валидации на страницах, содержащих несколько форм:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input
    id="email"
    type="email"
    class="@error('email', 'login') is-invalid @enderror"
/>

@error('email', 'login')
<div class="alert alert-danger">{{ $message }}</div>
@enderror
```

## # Стеки

Blade позволяет вам добавлять содержимое к именованным стекам, которые можно отобразить где-нибудь еще в другом шаблоне или макете. Это может быть особенно полезно для указания любых библиотек JavaScript, необходимых для ваших дочерних шаблонов:

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

Если вы хотите добавить содержимое с помощью директивы `@push`, только в том случае, если заданное логическое выражение возвращает `true`, вы можете использовать директиву `@pushIf`. Ниже приведен пример использования:

```
@pushIf($shouldPush, 'scripts')
<script src="/example.js"></script>
@endPushIf
```

Вы можете помещать в стек сколько угодно раз. Чтобы отобразить полное содержимое стека, передайте имя стека в директиву `@stack`:

```
<head>
  <!-- Head Contents -->

  @stack('scripts')
</head>
```

Если вы хотите добавить содержимое в начало стека, вы должны использовать директиву `@prepend`:

```
@push('scripts')
  This will be second...
@endpush

// Later...

@prepend('scripts')
  This will be first...
@endprepend
```

## # Внедрение служб

Директива `@inject` используется для извлечения службы из [контейнера служб Laravel](#). Первый аргумент, переданный в `@inject`, – это имя переменной, в которую будет помещена служба. Второй аргумент – это имя класса или интерфейса службы, которую вы хотите извлечь:

```
@inject('metrics', 'App\services\MetricsService')
```

```
<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

## # Рендеринг шаблонов Blade из строки

Иногда вам может потребоваться преобразовать сырую строку шаблона Blade в допустимый HTML. Вы можете сделать это, используя метод `render`, предоставляемый фасадом `Blade`. Метод `render` принимает строку шаблона Blade и необязательный массив данных, которые будут переданы в шаблон:

```
use Illuminate\Support\Facades\Blade;

return Blade::render('Hello, {{ $name }}', ['name' => 'Julian Bashir']);
```

Laravel рендерит встроенные шаблоны Blade, записывая их в директорию `storage/framework/views`. Если вы хотите, чтобы Laravel удалял эти временные файлы после рендеринга шаблона Blade, вы можете передать аргумент `deleteCachedView` методу:

```
return Blade::render(
    'Hello, {{ $name }}',
    ['name' => 'Julian Bashir'],
    deleteCachedView: true
);
```

## # Рендеринг фрагментов Blade

При использовании фронтенд фреймворков, таких как `Turbo` и `htmx`, иногда вам может потребоваться вернуть только часть шаблона Blade в HTTP-ответе. Фрагменты Blade позволяют вам сделать это. Для начала поместите часть вашего шаблона Blade между директивами `@fragment` и `@endfragment`:

```
@fragment('user-list')
<ul>
    @foreach ($users as $user)
        <li>{{ $user->name }}</li>
    @endforeach
```

```
</ul>
@endfragment
```

Затем, при рендеринге представления, которое использует этот шаблон, вы можете вызвать метод `fragment`, чтобы указать, что в исходящем HTTP-ответе должен быть включен только указанный фрагмент:

```
return view('dashboard', ['users' => $users])->fragment('user-list');
```

Метод `fragmentIf` позволяет условно возвращать фрагмент представления на основе заданного условия. В противном случае будет возвращено целое представление:

```
return view('dashboard', ['users' => $users])
->fragmentIf($request->hasHeader('HX-Request'), 'user-list');
```

Методы `fragments` и `fragmentsIf` позволяют возвращать несколько фрагментов представления в ответе. Фрагменты будут объединены в одну строку:

```
view('dashboard', ['users' => $users])
->fragments(['user-list', 'comment-list']);

view('dashboard', ['users' => $users])
->fragmentsIf(
    $request->hasHeader('HX-Request'),
    ['user-list', 'comment-list']
);
```

## # Расширение Blade

Blade позволяет вам определять ваши собственные пользовательские директивы с помощью метода `directive`. Когда компилятор Blade встречает вашу директиву, он вызывает указанное замыкание с выражением, содержащимся в директиве.

В следующем примере создается директива `@datetime($var)`, которая форматирует переданный `$var`, который должен быть экземпляром `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Blade::directive('datetime', function (string $expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

Как видите, мы привяжем метод `format` к любому выражению, переданному в директиву. Итак, в этом примере окончательный PHP, сгенерированный этой директивой, будет:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

После обновления логики директивы Blade вам нужно будет удалить все кешированные шаблоны Blade. Кешированные шаблоны Blade могут быть удалены с помощью команды `view:clear` Artisan.

## Пользовательские обработчики вывода

Если вы попытаетесь вывести объект при помощи Blade, у объекта будет вызван метод `__toString`. Метод `__toString` является одним из встроенных “магических методов” PHP. Однако иногда вы не можете контролировать метод `__toString` данного класса, например, когда класс, с которым вы взаимодействуете, принадлежит сторонней библиотеке.

В этих случаях Blade позволяет зарегистрировать пользовательский обработчик вывода для данного типа объектов. Для этого необходимо вызвать метод Blade `stringable`. Метод `stringable` принимает функцию, которая в аргументе принимает тип объекта, за рендеринг которого она отвечает. Обычно метод `stringable` следует вызывать в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Blade;
use Money\Money;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Blade::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

Теперь вы можете просто “вывести” экземпляр класса `Money` в шаблоне:

```
Cost: {{ $money }}
```

## Пользовательские операторы If

Программирование пользовательской директивы иногда бывает более сложным, чем необходимо при определении простых пользовательских условных операторов. По этой причине Blade содержит метод `Blade::if`, который позволяет быстро определять пользовательские условные директивы с помощью замыканий. Например, давайте определим условие, которое проверяет настроенный по умолчанию «диск» приложения. Мы можем сделать это в методе `boot` нашего `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

/**
 * Загрузка любых служб приложения.
 */
public function boot(): void
{
    Blade::if('disk', function (string $value) {
        return config('filesystems.default') === $value;
    });
}
```

После того как пользовательское условие было определено, вы можете использовать его в своих шаблонах:

```
@disk('local')
    <!-- The application is using the local disk... -->
@elsedisk('s3')
    <!-- The application is using the s3 disk... -->
@else
    <!-- The application is using some other disk... -->
@enddisk

@unlessdisk('local')
    <!-- The application is not using the local disk... -->
@enddisk
```

# Сборка ресурсов (Vite)

## # Введение

## # Установка и настройка

# Установка Node

# Установка Vite и плагина Laravel

# Настройка Vite

# Загрузка ваших скриптов и стилей

## # Запуск Vite

## # Работа с JavaScript

# Псевдонимы

# Vue

# React

# Inertia

# Обработка URL

## # Working With Stylesheets

## # Работа с Blade и маршрутами

# Обработка статических ресурсов с Vite

# Обновление при сохранении

# Псевдонимы

## # Предварительная выборка активов

## # Пользовательские базовые URL

## # Переменные среды

## # Отключение Vite в тестах

## # Рендеринг на стороне сервера (SSR)

## # Атрибуты тегов Style и Script

# Content Security Policy (CSP)Nonce

# Subresource Integrity (SRI) (Верность подресурсов)

# Произвольные атрибуты

## # Расширенная настройка

## # Введение

[Vite](#) – это современный инструмент сборки фронтенда, который обеспечивает чрезвычайно быстрое окружение разработки и собирает ваш код для продакшена. При создании приложений с использованием Laravel вы обычно используете Vite для сборки файлов CSS и JavaScript вашего приложения в готовые к продакшенну ресурсы.

Laravel интегрируется с Vite без проблем, предоставляя официальный плагин и директиву Blade для загрузки ваших ресурсов как для разработки, так и для продакшена.

Вы используете Laravel Mix? Vite заменил Laravel Mix в новых установках Laravel. Для документации по Mix посетите веб-сайт [Laravel Mix](#). Если вы хотите перейти на Vite, ознакомьтесь с нашим [руководством по миграции](#).

## Выбор между Vite и Laravel Mix

Прежде чем перейти на Vite, новые приложения Laravel использовали [Mix](#) при сборке ресурсов, который работает на основе [webpack](#). Vite сосредоточен на предоставлении более быстрого и продуктивного опыта при создании мощных JavaScript-приложений. Если вы разрабатываете одностраничное приложение (SPA), включая те, которые разработаны с использованием инструментов, таких как [Inertia](#), то Vite будет идеальным выбором.

Vite также хорошо работает с традиционными приложениями с серверным рендерингом с “примесью” JavaScript, включая те, которые используют [Livewire](#). Однако у него нет некоторых функций, которые поддерживает Laravel Mix, таких как возможность копирования произвольных ресурсов, на которые нет прямых ссылок в вашем приложении JavaScript, в сборку.

## Возвращение к Mix

Вы начали новое приложение Laravel, используя нашу структуру Vite, но вам нужно вернуться к Laravel Mix и webpack? Нет проблем. Пожалуйста, обратитесь к нашему [официальному руководству по миграции с Vite на Mix](#).

## # Установка и настройка

В следующей документации рассматривается процесс ручной установки и настройки плагина Laravel Vite. Однако стартовые комплекты Laravel уже включают в себя всю эту структуру и являются самым быстрым способом начать работу с Laravel и Vite.

### Установка Node

Перед запуском Vite и плагина Laravel убедитесь, что у вас установлены Node.js (версии 16 и выше) и NPM:

```
node -v  
npm -v
```

Вы можете легко установить последнюю версию Node и NPM, используя простые установщики с [официального сайта Node](#). Или, если вы используете [Laravel Sail](#), вы можете вызвать Node и NPM через Sail:

```
./vendor/bin/sail node -v  
./vendor/bin/sail npm -v
```

### Установка Vite и плагина Laravel

В новой установке Laravel в корне структуры каталогов вашего приложения вы найдете файл `package.json`. В файле `package.json` уже содержится все необходимое для начала работы с Vite и плагином Laravel. Вы можете установить зависимости фронтенда вашего приложения через NPM:

```
npm install
```

## Настройка Vite

Vite настраивается с помощью файла `vite.config.js` в корне вашего проекта. Вы можете настраивать этот файл по своему усмотрению, а также устанавливать любые другие плагины, необходимые для вашего приложения, такие как `@vitejs/plugin-vue` или `@vitejs/plugin-react`.

Плагин Laravel Vite требует указания точек входа для вашего приложения. Это могут быть файлы JavaScript или CSS, включая предварительно обработанные языки, такие как TypeScript, JSX, TSX и Sass.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel([
      'resources/css/app.css',
      'resources/js/app.js',
    ]),
  ],
});
```

Если вы создаете SPA, включая приложения, построенные с использованием Inertia, то Vite будет лучше всего работать без точек входа CSS:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel([
      'resources/css/app.css', // [t1! remove]
      'resources/js/app.js',
    ]),
  ],
});
```

Вместо этого вы должны импортировать свой CSS через JavaScript. Обычно это делается в файле `resources/js/app.js` вашего приложения:

```
import './bootstrap';
import '../css/app.css'; // [t1! add]
```

Плагин Laravel также поддерживает несколько точек входа и расширенные параметры конфигурации, такие как [точки входа SSR](#).

## Работа с защищенным сервером разработки

Если ваш локальный веб-сервер разработки обслуживает ваше приложение через HTTPS, у вас могут возникнуть проблемы с подключением к серверу разработки Vite.

Если вы используете [Laravel Herd](#) и зашифровали сайт, или вы используете [Laravel Valet](#) и запустили [команду secure](#) для вашего приложения, плагин Laravel Vite автоматически обнаружит и использует сгенерированный TLS-сертификат.

Если вы зашифровали сайт с использованием хоста, который не соответствует имени каталога приложения, вы можете вручную указать хост в файле `vite.config.js` вашего приложения:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      detectTls: 'my-app.test', // [t1! add]
    }),
  ],
});
```

Когда вы используете другой веб-сервер, вы должны сгенерировать доверенный сертификат и вручную настроить Vite для использования сгенерированных сертификатов:

```
// ...
import fs from 'fs'; // [tl! add]

const host = 'my-app.test'; // [tl! add]

export default defineConfig({
  // ...
  server: { // [tl! add]
    host, // [tl! add]
    hmr: { host }, // [tl! add]
    https: { // [tl! add]
      key: fs.readFileSync(`/path/to/${host}.key`), // [tl! add]
      cert: fs.readFileSync(`/path/to/${host}.crt`), // [tl! add]
    }, // [tl! add]
  }, // [tl! add]
});
```

Если вы не можете сгенерировать доверенный сертификат для вашей системы, вы можете установить и настроить плагин [@vitejs/plugin-basic-ssl](#). При использовании ненадежных сертификатов вам нужно будет принять предупреждение о сертификате для сервера разработки Vite в вашем браузере, перейдя по ссылке "Local" в консоли при выполнении команды `npm run dev`.

## Запуск сервера разработки в Sail на WSL2

При запуске сервера разработки Vite в [Laravel Sail](#) на Windows Subsystem for Linux 2 (WSL2), вам следует добавить следующую конфигурацию в ваш файл `vite.config.js`, чтобы обеспечить связь браузера с сервером разработки:

```
// ...

export default defineConfig({
  // ...
  server: { // [tl! add:start]
    hmr: {
      host: 'localhost',
    },
  }, // [tl! add:end]
});
```

## Загрузка ваших скриптов и стилей

После настройки точек входа Vite вы можете ссылаться на них с помощью директивы `@vite()` в Blade, которую вы должны добавить в `<head>` корневого шаблона вашего приложения:

```
<!DOCTYPE html>
<head>
{{-- ... --}}
@vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

Если вы импортируете свой CSS через JavaScript, вам нужно включить только точку входа JavaScript:

```
<!DOCTYPE html>
<head>
{{-- ... --}}
@vite('resources/js/app.js')
</head>
```

Директива `@vite` автоматически обнаруживает сервер разработки Vite и внедряет клиент Vite для возможности горячей замены модулей. В режиме сборки директива загрузит ваши скомпилированные и пронумерованные ресурсы, включая любой импортированный CSS.

При необходимости вы также можете указать путь сборки ваших скомпилированных ресурсов при вызове директивы `@vite`.

```
<!doctype html>
<head>
{{-- Given build path is relative to public path. --}}
@vite('resources/js/app.js', 'vendor/courier/build')
</head>
```

## Встраивание ресурсов

Иногда может быть необходимо включить сырое содержимое ресурсов, а не ссылаться на версионный URL ресурса. Например, вам может понадобиться

включить содержимое ресурса непосредственно на страницу, когда передаете HTML-контент генератору PDF. Вы можете выводить содержимое ресурсов Vite с помощью метода `content`, предоставленного фасадом `Vite`:

```
@use('Illuminate\Support\Facades\Vite')  
  
<!doctype html>  
<head>  
    {{-- ... --}}  
  
    <style>  
        {!! Vite::content('resources/css/app.css') !!}  
    </style>  
    <script>  
        {!! Vite::content('resources/js/app.js') !!}  
    </script>  
</head>
```

## # Запуск Vite

Существует два способа запуска Vite. Вы можете запустить сервер разработки с помощью команды `dev`, что полезно во время локальной разработки. Сервер разработки автоматически обнаруживает изменения в ваших файлах и мгновенно отображает их в любых открытых окнах браузера.

Или выполнить команду `build`, которая версионирует и собирает ресурсы вашего приложения, подготовив их к развертыванию в производственной среде:

```
# Run the Vite development server...  
npm run dev  
  
# Build and version the assets for production...  
npm run build
```

Если вы запускаете сервер разработки в `Sail` на WSL2, вам могут потребоваться дополнительные [опции конфигурации](#).

## # Работа с JavaScript

## Псевдонимы

По умолчанию плагин Laravel предоставляет общий псевдоним, чтобы вы могли начать работу сразу и удобно импортировать ресурсы вашего приложения:

```
{  
    '@' => '/resources/js'  
}
```

Вы можете переопределить псевдоним '`@`', добавив собственный в файл конфигурации `vite.config.js`:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
  
export default defineConfig({  
    plugins: [  
        laravel(['resources/ts/app.tsx']),  
    ],  
    resolve: {  
        alias: {  
            '@': '/resources/ts',  
        },  
    },  
});
```

## Vue

Если вы хотите собрать свой фронтенд, используя фреймворк [Vue](#), вам также нужно установить плагин `@vitejs/plugin-vue`:

```
npm install --save-dev @vitejs/plugin-vue
```

Затем вы можете включить плагин в ваш файл конфигурации `vite.config.js`.

При использовании плагина Vue с Laravel вам потребуются несколько дополнительных параметров:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
import vue from '@vitejs/plugin-vue';
```

```
export default defineConfig({
  plugins: [
    laravel(['resources/js/app.js']),
    vue({
      template: {
        transformAssetUrls: {
          // Плагин Vue перепишет URL-адреса ресурсов, когда они
          // будут использоваться в однофайловых компонентах,
          // чтобы указывать на веб-сервер Laravel. Установка этого
          // значения в `null` позволяет вместо этого плагину Laravel
          // переписывать URL-адреса ресурсов, чтобы они указывали на сервер
          base: null,
          // Плагин Vue будет анализировать абсолютные URL-адреса и рассматри-
          // вать их как абсолютные пути к файлам на диске. Установка этого значе-
          //ния в `false` оставит абсолютные URL-адреса нетронутыми, чтобы они не
          // ссылаться на ресурсы в папке public, как ожидается.
          includeAbsolute: false,
        },
      },
    }),
  ],
});
```

Стартовые наборы Laravel ([starter kits](#)) уже включают правильную конфигурацию Laravel, Vue и Vite. Посмотрите [Laravel Breeze](#) для самого быстрого способа начать работу с Laravel, Vue и Vite.

## React

Если вы хотите собрать свой фронтенд, используя фреймворк [React](#), вам также необходимо установить плагин [@vitejs/plugin-react](#):

```
npm install --save-dev @vitejs/plugin-react
```

Затем вы можете включить плагин в ваш файл конфигурации [vite.config.js](#):

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [
    laravel(['resources/js/app.jsx']),
    react(),
  ],
});
```

Вы должны убедиться, что любые файлы, содержащие JSX, имеют расширение `.jsx` или `.tsx`, помня о необходимости обновления вашей точки входа, если это требуется, как показано [выше](#).

Вам также потребуется включить дополнительную директиву `@viteReactRefresh` вместе с вашей текущей директивой `@vite` в Blade.

```
@viteReactRefresh
@vite('resources/js/app.jsx')
```

Директива `@viteReactRefresh` должна быть вызвана перед директивой `@vite`.

Стартовые наборы Laravel ([starter kits](#)) уже включают правильную конфигурацию Laravel, React и Vite. Ознакомьтесь с [Laravel Breeze](#) – самым быстрым способом начать работу с Laravel, React и Vite.

## Inertia

Плагин Laravel Vite предоставляет удобную функцию `resolvePageComponent`, которая поможет вам определить ваши компоненты страниц Inertia. Ниже приведен пример использования помощника с Vue 3; однако, вы также можете использовать эту функцию в других фреймворках, таких как React:

```
import { createApp, h } from 'vue';
import { createInertiaApp } from '@inertiajs/vue3';
import { resolvePageComponent } from 'laravel-vite-plugin/inertia-helpers';

createInertiaApp({
    resolve: (name) => resolvePageComponent(`./Pages/${name}.vue`, import.meta.glob(`..
    setup({ el, App, props, plugin }) {
        createApp({ render: () => h(App, props) })
            .use(plugin)
            .mount(el)
    },
});
```

Если вы используете функцию разделения кода Vite с Inertia, мы рекомендуем настроить [предварительную выборку актива](#).

Стартовые наборы Laravel ([starter kits](#)) уже включают правильную конфигурацию Laravel, Inertia и Vite. Ознакомьтесь с [Laravel Breeze](#) чтобы быстро начать работу с Laravel, Inertia и Vite.

## Обработка URL

При использовании Vite и ссылок на ресурсы в HTML, CSS или JS вашего приложения, следует учитывать несколько моментов. Во-первых, если вы ссылаетесь на ресурсы с абсолютным путем, Vite не включит ресурс в сборку; поэтому убедитесь, что ресурс доступен в вашей публичной директории. Вам следует избегать использования абсолютных путей при использовании [выделенной точки входа CSS](#), поскольку во время разработки браузеры будут пытаться загрузить эти пути с сервера разработки Vite, где размещен CSS, а не из вашего общедоступного каталога.

При использовании относительных путей к ресурсам помните, что пути относительны файлу, в котором они используются. Любые ресурсы, ссылки на которые осуществляются через относительный путь, будут переписаны, версионированы и собраны Vite.

Рассмотрим следующую структуру проекта:

```
public/
  taylor.png
resources/
  js/
    Pages/
      Welcome.vue
  images/
    abigail.png
```

Следующий пример демонстрирует, как Vite будет обрабатывать относительные и абсолютные URL-адреса:

```
<!-- Этот ресурс не обрабатывается Vite и не будет включен в сборку. -->


<!-- Этот ресурс будет переписан, пронумерован и собран Vite. -->

```

## # Working With Stylesheets

Вы можете узнать больше о поддержке CSS в Vite в [документации Vite](#). Если вы используете плагины PostCSS, такие как [Tailwind](#), вы можете создать файл `postcss.config.js` в корне вашего проекта, и Vite автоматически его применит.

```
export default {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
};
```

Стартовые наборы Laravel ([starter kits](#)) уже включают правильную конфигурацию Tailwind, PostCSS и Vite. Или, если вы хотите использовать Tailwind и Laravel без использования одного из наших стартовых наборов, ознакомьтесь с [руководством по установке Tailwind для Laravel](#).

# # Работа с Blade и маршрутами

## Обработка статических ресурсов с Vite

При ссылке на ресурсы в вашем JavaScript или CSS, Vite автоматически обрабатывает и версионирует их. Кроме того, при построении приложений на основе Blade, Vite также может обрабатывать и версионировать статические ресурсы, на которые вы ссылаетесь исключительно в шаблонах Blade.

Однако для этого необходимо осведомить Vite о ваших ресурсах, импортировав статические ресурсы в точку входа вашего приложения. Например, если вы хотите обработать и версионировать все изображения, хранящиеся в `resources/images`, и все шрифты, хранящиеся в `resources/fonts`, вам следует добавить следующее в точку входа вашего приложения `resources/js/app.js`:

```
import.meta.glob([
  './images/**',
  './fonts/**',
]);
```

Теперь эти ресурсы будут обрабатываться Vite при запуске `npm run build`. Затем вы можете ссылаться на эти ресурсы в шаблонах Blade, используя метод `Vite::asset`, который вернет пронумерованный URL для указанного ресурса:

```

```

## Обновление при сохранении

Когда ваше приложение построено с использованием традиционного рендеринга на стороне сервера с помощью Blade, Vite может улучшить ваш рабочий процесс разработки, автоматически обновляя браузер при внесении изменений в файлы представлений вашего приложения. Чтобы начать, просто укажите параметр `refresh` как `true`.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
```

```
    laravel({
        // ...
        refresh: true,
    }),
],
});
```

Когда параметр `refresh` установлен в `true`, сохранение файлов в следующих каталогах вызовет полное обновление страницы в браузере при выполнении команды `npm run dev`:

- `app/Livewire/**`
- `app/View/Components/**`
- `lang/**`
- `resources/lang/**`
- `resources/views/**`
- `routes/**`

Отслеживание каталога `routes/**` полезно, если вы используете [Ziggy](#) для создания ссылок на маршруты во фронтенде вашего приложения.

Если эти стандартные пути не соответствуют вашим потребностям, вы можете указать собственный список путей для отслеживания:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: ['resources/views/**'],
        }),
    ],
});
```

В основе плагина Laravel Vite используется пакет [vite-plugin-full-reload](#), который предлагает некоторые дополнительные параметры конфигурации для настройки

поведения этой функции. Если вам нужен такой уровень настройки, вы можете предоставить определение `config`:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: [
                paths: ['path/to/watch/**'],
                config: { delay: 300 }
            ],
        }),
    ],
});
```

## Псевдонимы

Часто в JavaScript-приложениях создают [псевдонимы](#) для часто используемых каталогов. Однако, вы также можете создавать псевдонимы для использования в Blade, используя метод `macro` класса `Illuminate\Support\Facades\Vite`. Обычно “макросы” определяются в методе `boot` сервис-провайдера:

```
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Vite::macro('image', fn (string $asset) => $this->asset("resources/images/{$asset}")
}
```

После определения макроса его можно вызвать в ваших шаблонах. Например, мы можем использовать определенный выше макрос `image`, чтобы ссылаться на ресурс, расположенный по пути `resources/images/logo.png`:

```

```

## # Предварительная выборка активов

При создании SPA с использованием функции разделения кода Vite необходимые ресурсы извлекаются при навигации по каждой странице. Такое поведение может привести к задержке рендеринга пользовательского интерфейса. Если это проблема для выбранной вами среды внешнего интерфейса, Laravel предлагает возможность предварительной загрузки ресурсов JavaScript и CSS вашего приложения при начальной загрузке страницы.

Вы можете поручить Laravel выполнить предварительную выборку ваших ресурсов, вызвав метод `Vite::prefetch` в методе [boot поставщика услуг](#):

```
<?php
namespace App\Providers;
use Illuminate\Support\Facades\Vite;
use Illuminate\Support\ServiceProvider;
class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        // ...
    }
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Vite::prefetch(concurrency: 3);
    }
}
```

В приведенном выше примере ресурсы будут предварительно загружены с максимум `3` одновременными загрузками при каждой загрузке страницы. Вы можете изменить параллелизм в соответствии с потребностями вашего приложения или не указывать ограничение параллелизма, если приложение должно загружать все ресурсы одновременно:

```
/**
 * Bootstrap any application services.
 */
```

```
public function boot(): void
{
    Vite::prefetch();
}
```

По умолчанию предварительная выборка начинается при возникновении события [load](#). Если вы хотите настроить время начала предварительной загрузки, вы можете указать событие, которое Vite будет прослушивать.

```
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Vite::prefetch(event: 'vite:prefetch');
}
```

Учитывая приведенный выше код, предварительная выборка теперь начнется, когда вы вручную отправите событие `vite:prefetch` для объекта `window`. Например, вы можете начать предварительную загрузку через три секунды после загрузки страницы:

```
<script>
    addEventListener('load', () => setTimeout(() => {
        dispatchEvent(new Event('vite:prefetch'))
    }, 3000))
</script>
```

## # Пользовательские базовые URL

Если ваши скомпилированные ресурсы Vite развернуты на домене, отличном от вашего приложения, например, через CDN, вы должны указать переменную окружения `ASSET_URL` в файле `.env` вашего приложения:

```
ASSET_URL=https://cdn.example.com
```

После настройки URL ресурса в начало всех переписанных URL-адреса ваших ресурсов будет добавлено указанное значение:

<https://cdn.example.com/build/assets/app.9dce8d17.js>

Помните, что абсолютные URL-адреса не переписываются Vite, поэтому они не будут изменены.

## # Переменные среды

Вы можете внедрить переменные среды в ваш JavaScript, добавив им префикс `VITE_` в файле `.env` вашего приложения:

```
VITE_SENTRY_DSN_PUBLIC=http://example.com
```

Вы можете получить доступ к внедренным переменным среды через объект `import.meta.env`:

```
import.meta.env.VITE_SENTRY_DSN_PUBLIC
```

## # Отключение Vite в тестах

Интеграция Vite в Laravel будет пытаться разрешить ваши ресурсы во время выполнения ваших тестов, что требует запуска сервера разработки Vite или сборки ваших ресурсов.

Если вы предпочитаете использовать имитацию Vite во время тестирования, вы можете вызвать метод `withoutVite`, который доступен для всех тестов, расширяющих класс `TestCase` Laravel:

Pest      PHPUnit

```
test('without vite example', function () {
    $this->withoutVite();
    // ...
});
```

Если вы хотите отключить Vite для всех тестов, вы можете вызвать метод `withoutVite` из метода `setUp` вашего базового класса `TestCase`:

```
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    protected function setUp(): void// [tl! add:start]
    {
        parent::setUp();

        $this->withoutVite();
    }// [tl! add:end]
}
```

## # Рендеринг на стороне сервера (SSR)

Плагин Laravel Vite облегчает настройку рендеринга на стороне сервера с использованием Vite. Чтобы начать, создайте точку входа SSR в `resources/js/ssr.js` и укажите ее в плагине Laravel, передав конфигурационную опцию:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            input: 'resources/js/app.js',
            ssr: 'resources/js/ssr.js',
        }),
    ],
});
```

Чтобы не забыть перестроить точку входа SSR, мы рекомендуем изменить скрипт "build" в файле `package.json` вашего приложения для создания сборки SSR:

```
"scripts": {
    "dev": "vite",
```

```
"build": "vite build" // [tl! remove]
"build": "vite build && vite build --ssr" // [tl! add]
}
```

Затем, чтобы собрать и запустить сервер SSR, вы можете выполнить следующие команды:

```
npm run build
node bootstrap/ssr/ssr.js
```

Если вы используете [SSR с Inertia](#), вы можете вместо этого использовать команду Artisan `inertia:start-ssr` для запуска сервера SSR:

```
php artisan inertia:start-ssr
```

Стартовые наборы Laravel ([starter kits](#)) уже включают правильную конфигурацию Laravel, SSR Inertia и Vite. Ознакомьтесь с [Laravel Breeze](#) для самого быстрого способа начать работу с Laravel, SSR Inertia и Vite.

## # Атрибуты тегов Style и Script

### Content Security Policy (CSP) Nonce

Если вы хотите включить атрибут `nonce` в ваших тегах `script` и `style` как часть [политики безопасности контента \(Content Security Policy\)](#), вы можете сгенерировать или указать nonce, используя метод `useCspNonce` внутри собственного [middleware](#):

```
<?php

namespace App\Http\Middleware;

use Closure;
```

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Vite;
use Symfony\Component\HttpFoundation\Response;

class AddContentSecurityPolicyHeaders
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        Vite::useCspNonce();

        return $next($request)->withHeaders([
            'Content-Security-Policy' => "script-src 'nonce-".Vite::cspNonce()."",
        ]);
    }
}
```

После вызова метода `useCspNonce`, Laravel автоматически включит атрибуты `nonce` во все сгенерированные теги `script` и `style`.

Если вам нужно указать nonce в другом месте, включая [директиву `@route Ziggy`](#), входящую в стартовые наборы Laravel, вы можете сделать это, используя метод `cspNonce`:

```
@routes(nonce: Vite::cspNonce())
```

Если у вас уже есть nonce, который вы хотели бы использовать, вы можете передать его методу `useCspNonce`:

```
Vite::useCspNonce($nonce);
```

## Subresource Integrity (SRI) (Велостность подресурсов)

Если ваш манифест Vite включает хеши `integrity` для ваших ресурсов, Laravel автоматически добавит атрибут `integrity` ко всем тегам `script` и `style`, которые он

генерирует, чтобы обеспечить [целостность подресурсов](#). По умолчанию Vite не включает хеш `integrity` в свой манифест, но вы можете включить его, установив плагин [vite-plugin-manifest-sri](#) из NPM:

```
npm install --save-dev vite-plugin-manifest-sri
```

Затем вы можете включить этот плагин в вашем файле `vite.config.js`:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import manifestSRI from 'vite-plugin-manifest-sri';// [tl! add]

export default defineConfig({
  plugins: [
    laravel({
      // ...
    }),
    manifestSRI(),// [tl! add]
  ],
});
```

При необходимости вы также можете настроить ключ манифеста, где будет находиться хеш целостности:

```
use Illuminate\Support\Facades\Vite;

Vite::useIntegrityKey('custom-integrity-key');
```

Если вы хотите полностью отключить эту автообнаружение, вы можете передать `false` методу `useIntegrityKey`:

```
Vite::useIntegrityKey(false);
```

## Произвольные атрибуты

Если вам нужно добавить дополнительные атрибуты к вашим тегам `script` и `style`, такие как атрибут [data-turbo-track](#), вы можете указать их с помощью методов

`useScriptTagAttributes` И `useStyleTagAttributes`. Обычно эти методы вызываются из [сервис-провайдера](#):

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes([
    'data-turbo-track' => 'reload', // Specify a value for the attribute...
    'async' => true, // Specify an attribute without a value...
    'integrity' => false, // Exclude an attribute that would otherwise be included...
]);

Vite::useStyleTagAttributes([
    'data-turbo-track' => 'reload',
]);
```

Если вам нужно использовать условие для добавления атрибутов, вы можете передать обратный вызов, который будет получать исходный путь ресурса, его URL, его фрагмент манифеста и весь манифест:

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes(fn (string $src, string $url, array|null $chunk, array|n
    'data-turbo-track' => $src === 'resources/js/app.js' ? 'reload' : false,
]);

Vite::useStyleTagAttributes(fn (string $src, string $url, array|null $chunk, array|n
    'data-turbo-track' => $chunk && $chunk['isEntry'] ? 'reload' : false,
});
```

Аргументы `$chunk` И `$manifest` будут равны `null`, если сервер разработки Vite запущен.

## # Расширенная настройка

По умолчанию плагин Vite Laravel использует соглашения, которые должны подходить для большинства приложений; однако иногда вам может потребоваться настроить поведение Vite. Для активации дополнительных параметров настройки

мы предлагаем следующие методы и параметры, которые могут использоваться вместо директивы Blade `@vite`:

```
<!doctype html>
<head>
{{-- ... --}}


{{
    Vite::useHotFile(storage_path('vite.hot')) // Customize the "hot" file...
    ->useBuildDirectory('bundle') // Customize the build directory...
    ->useManifestFilename('assets.json') // Customize the manifest filename.
    ->withEntryPoints(['resources/js/app.js']) // Specify the entry points...
    ->createAssetPathsUsing(function (string $path, ?bool $secure) { // Cust...
        return "https://cdn.example.com/{$path}";
    })
}}
</head>
```

Затем в файле `vite.config.js` вы должны указать ту же конфигурацию:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            hotFile: 'storage/vite.hot', // Customize the "hot" file...
            buildDirectory: 'bundle', // Customize the build directory...
            input: ['resources/js/app.js'], // Specify the entry points...
        }),
    ],
    build: {
        manifest: 'assets.json', // Customize the manifest filename...
    },
});
```

## Коррекция URL-адресов сервера разработки

Некоторые плагины в экосистеме Vite предполагают, что URL-адреса, начинающиеся с косой черты, всегда будут указывать на сервер разработки Vite. Однако из-за характера интеграции с Laravel это не так.

Например, плагин `vite-imagetools` выводит URL-адреса, подобные следующим, пока Vite обслуживает ваши ресурсы:

```

```

Плагин `vite-imagetools` ожидает, что выходной URL будет перехвачен Vite, и затем плагин сможет обрабатывать все URL-адреса, которые начинаются с `/@imagedtools`. Если вы используете плагины, ожидающие такое поведение, вам придется вручную скорректировать URL-адреса. Вы можете сделать это в вашем файле `vite.config.js`, используя опцию `transformOnServe`.

В этом конкретном примере мы добавим префикс URL сервера разработки ко всем вхождениям `/@imagedtools` в сгенерированном коде:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import { imagedtools } from 'vite-imagedtools';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      transformOnServe: (code, devServerUrl) => code.replaceAll('/@imagedtools',
    }),
    imagedtools(),
  ],
});
```

Теперь, когда Vite обслуживает ресурсы, он будет выводить URL-адреса, указывающие на сервер разработки Vite:

```
- <!-- [t1! remove]
+ 
```

# Генерация URL-адресов

# Введение

# Основы

# Создание URL

# Доступ к текущему URL

# URL для именованных маршрутов

# Подписанные URL

# URL для действий контроллера

# Значения по умолчанию

## # Введение

Laravel предлагает несколько функций, которые помогут вам в создании URL-адресов для вашего приложения. Эти помощники в первую очередь полезны при построении ссылок в ваших шаблонах и ответах API или при создании ответов-перенаправлений в другую часть вашего приложения.

## # Основы

### Создание URL

Помощник `url` используется для генерации произвольных URL-адресов для вашего приложения. Сгенерированный URL-адрес будет автоматически использовать схему (HTTP или HTTPS) и хост из текущего запроса, обрабатываемого приложением:

```
$post = App\Models\Post::find(1);  
  
echo url("/posts/{$post->id}");  
  
// http://example.com/posts/1
```

Чтобы сгенерировать URL-адрес с параметрами строки запроса, вы можете использовать метод `query`:

```
echo url()->query('/posts', ['search' => 'Laravel']);  
  
// https://example.com/posts?search=Laravel  
  
echo url()->query('/posts?sort=latest', ['search' => 'Laravel']);  
  
// http://example.com/posts?sort=latest&search=Laravel
```

Предоставление параметров строки запроса, которые уже существуют в адресе, перезапишет их существующее значение:

```
echo url()->query('/posts?sort=latest', ['sort' => 'oldest']);  
  
// http://example.com/posts?sort=oldest
```

Массивы значений также могут передаваться в качестве параметров запроса. Эти значения будут правильно введены и закодированы в сгенерированном URL-адресе:

```
echo $url = url()->query('/posts', ['columns' => ['title', 'body']]);  
  
// http://example.com/posts?columns%5B0%5D=title&columns%5B1%5D=body  
  
echo urldecode($url);  
  
// http://example.com/posts?columns[0]=title&columns[1]=body
```

## Доступ к текущему URL

Если не передан путь помощнику `url`, то возвращается экземпляр `Illuminate\Routing\UrlGenerator`, позволяющий вам получить доступ к информации о текущем URL:

```
// Получить текущий URL без строки запроса ...  
echo url()->current();  
  
// Получить текущий URL, включая строку запроса ...
```

```
echo url()->full();  
  
// Получить полный URL-адрес предыдущего запроса ...  
echo url()->previous();
```

К каждому из этих методов также можно получить доступ через [фасад URL](#):

```
use Illuminate\Support\Facades\URL;  
  
echo URL::current();
```

## # URL для именованных маршрутов

Помощник `route` используется для генерации URL-адресов для [именованных маршрутов](#). Именованные маршруты позволяют создавать URL-адреса без привязки к фактическому URL-адресу, определенному в маршруте. Следовательно, если URL-адрес маршрута изменится, никаких изменений в ваши вызовы функции `route` вносить не нужно. Например, представьте, что ваше приложение содержит маршрут, определенный следующим образом:

```
Route::get('/post/{post}', function (Post $post) {  
    // ...  
})->name('post.show');
```

Чтобы сгенерировать URL-адрес этого маршрута, вы можете использовать помощник `route` следующим образом:

```
echo route('post.show', ['post' => 1]);  
  
// http://example.com/post/1
```

Конечно, помощник `route` также может использоваться для генерации URL-адресов для маршрутов с несколькими параметрами:

```
Route::get('/post/{post}/comment/{comment}', function (Post $post, Comment $comment)  
    // ...  
})->name('comment.show');
```

```
echo route('comment.show', ['post' => 1, 'comment' => 3]);  
  
// http://example.com/post/1/comment/3
```

Любые дополнительные элементы массива, не соответствующие параметрам определения маршрута, будут добавлены в строку запроса URL:



```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);  
  
// http://example.com/post/1?search=rocket
```

## Модели Eloquent

Вы часто будете генерировать URL-адреса, используя ключ маршрута (обычно первичный ключ) [модели Eloquent](#). По этой причине вы можете передавать модели Eloquent в качестве значений параметров. Помощник `route` автоматически извлечет ключ маршрута модели:

```
echo route('post.show', ['post' => $post]);
```

## Подписанные URL

Laravel позволяет вам легко создавать «подписанные» URL-адреса для именованных маршрутов. Эти URL-адреса имеют хеш «подписи», добавленный к строке запроса, который позволяет Laravel проверять, что URL-адрес не был изменен с момента его создания. Подписанные URL-адреса особенно полезны для маршрутов, которые общедоступны, но требуют уровня защиты от манипуляций с URL-адресами.

Например, вы можете использовать подписанные URL-адреса для реализации общедоступной ссылки «отказаться от подписки», которая отправляется вашим клиентам по электронной почте. Чтобы создать подписанный URL для именованного маршрута, используйте метод `signedRoute` фасада `URL`:

```
use Illuminate\Support\Facades\URL;  
  
return URL::signedRoute('unsubscribe', ['user' => 1]);
```

Вы можете исключить домен из хеша подписанного URL, предоставив аргумент `absolute` методу `signedRoute`:

```
return URL::signedRoute('unsubscribe', ['user' => 1], absolute: false);
```

Если вы хотите сгенерировать временный подписанный URL-адрес маршрута, срок действия которого истекает по истечении определенного времени, вы можете использовать метод `temporarySignedRoute`. Когда Laravel проверяет временный подписанный URL-адрес маршрута, он гарантирует, что метка времени истечения срока, закодированная в подписанный URL-адрес, не истекла:

```
use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

## Проверка запросов подписанного маршрута

Чтобы убедиться, что входящий запрос имеет действительную подпись, вы должны вызывать метод `hasValidSignature` для входящего объекта запроса `Illuminate\Http\Request`:

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (! $request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

Иногда может потребоваться разрешить фронтенду вашего приложения добавлять данные к подписенному URL, например, при выполнении пагинации на стороне клиента. Поэтому вы можете указать параметры запроса, которые следует игнорировать при проверке подписанного URL, используя метод `hasValidSignatureWhileIgnoring`. Помните, что игнорирование параметров позволяет любому изменять эти параметры в запросе:

```
if (! $request->hasValidSignatureWhileIgnoring(['page', 'order'])) {  
    abort(401);  
}
```

Вместо проверки подписанных URL-адресов с помощью экземпляра входящего запроса вы можете назначить маршруту `signed` ([Illuminate\Routing\Middleware\ValidateSignature](#)) [посредника \(middleware\)](#). Если входящий запрос не имеет действительной подписи, промежуточное программное обеспечение автоматически вернет HTTP-ответ «403»:

```
Route::post('/unsubscribe/{user}', function (Request $request) {  
    // ...  
})->name('unsubscribe')->middleware('signed');
```

Если ваши подписанные URL-адреса не включают домен в хеш URL, вы должны предоставить аргумент `relative` промежуточному программному обеспечению:

```
Route::post('/unsubscribe/{user}', function (Request $request) {  
    // ...  
})->name('unsubscribe')->middleware('signed:relative');
```

## Ответ на недействительные подписанные маршруты

Когда кто-то посещает подписанный URL-адрес, срок действия которого истек, он получит общую страницу с ошибкой для кода состояния `403` HTTP. Однако вы можете настроить это поведение, определив собственное замыкание «рендеринга» для исключения [InvalidSignatureException](#) в файле `bootstrap/app.php` вашего приложения:

```
use Illuminate\Routing\Exceptions\InvalidSignatureException;  
  
->withExceptions(function (Exceptions $exceptions) {  
    $exceptions->render(function (InvalidSignatureException $e) {  
        return response()->view('errors.link-expired', status: 403);  
    });  
})
```

## # URL для действий контроллера

Функция `action` генерирует URL-адрес для переданного действия контроллера:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

Если метод контроллера принимает параметры маршрута, вы можете передать ассоциативный массив параметров маршрута в качестве второго аргумента функции:

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

## # Значения по умолчанию

Для некоторых приложений вы можете указать значения по умолчанию для определенных параметров URL-адреса. Например, представьте, что многие из ваших маршрутов определяют параметр `{locale}`:

```
Route::get('/{locale}/posts', function () {
    // ...
})->name('post.index');
```

Обременительно передавать `locale` каждый раз при вызове помощника `route`. Итак, вы можете использовать метод `URL::defaults`, чтобы определить значение по умолчанию для этого параметра, которое всегда будет применяться во время текущего запроса. Вы можете вызвать этот метод из [посредника маршрута](#), чтобы получить доступ к текущему запросу:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;
use Symfony\Component\HttpFoundation\Response;

class SetDefaultLocaleForUrls
{
```

```

/**
 * Обработка входящего запроса.
 *
 * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
 */
public function handle(Request $request, Closure $next): Response
{
    URL::defaults(['locale' => $request->user()->locale]);

    return $next($request);
}
}

```

После установки значения по умолчанию для параметра `locale` вам больше не потребуется передавать его значение при генерации URL-адресов с помощью помощника `route`.

## Параметры URL по умолчанию и приоритет посредника

Установка значений URL по умолчанию может мешать Laravel обрабатывать неявные привязки модели. Следовательно, необходимо [установить приоритет посреднику](#), который задает значения URL по умолчанию, и должен выполняться перед посредником Laravel `SubstituteBindings`. Вы можете сделать это, используя метод промежуточного программного обеспечения `priority` в файле `bootstrap/app.php` вашего приложения:

```

->withMiddleware(function (Middleware $middleware) {
    $middleware->priority([
        \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,
        \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
        \Illuminate\Routing\Middleware\ThrottleRequests::class,
        \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
        \Illuminate\Session\Middleware\AuthenticateSession::class,
        \App\Http\Middleware\SetDefaultLocaleForUrls::class, // [tl! add]
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        \Illuminate\Auth\Middleware\Authorize::class,
    ]);
})

```

# Сессия HTTP

## # Введение

- # Конфигурирование
- # Предварительная подготовка драйверов

## # Взаимодействие с сессией

- # Получение данных
- # Получение части данных сессии
- # Сохранение данных
- # Кратковременные данные
- # Удаление данных
- # Пересоздание идентификатора сессии

## # Блокировка сессии

## # Добавление пользовательских драйверов сессии

## # Введение

Поскольку приложения, использующие HTTP, не имеют состояния, то сессии позволяют хранить пользовательскую информацию между несколькими запросами. Эта пользовательская информация обычно помещается в постоянное хранилище, к которому можно получить доступ из последующих запросов.

Laravel предлагает множество различных типов хранилищ сессий, доступ к которым осуществляется через выразительный унифицированный API. Осуществлена поддержка популярных типов хранилищ, таких как [Memcached](#), [Redis](#) и база данных.

## Конфигурирование

Конфигурационный файл сессии вашего приложения расположен в [config/session.php](#). Обязательно просмотрите параметры, доступные вам в этом файле. По умолчанию Laravel ориентирован на использование драйвера сессии [database](#).

Параметр конфигурации `driver` сессии определяет, где будут храниться данные сессии для каждого запроса. Laravel включает в себя множество драйверов:

- `file` – сессии хранятся в `storage/framework/sessions`.
- `cookie` – сессии хранятся в безопасных, зашифрованных файлах Cookies.
- `database` – сессии хранятся в реляционной базе данных.
- `memcached` / `redis` – сессии хранятся в одном из этих быстрых хранилищ на основе кеша.
- `dynamodb` – сессии хранятся в AWS DynamoDB.
- `array` – сессии хранятся в массиве PHP и не будут сохранены.

Драйвер `array` в основном используется во время [тестирования](#) и предотвращает сохранение данных, находящихся в сессии.

## Предварительная подготовка драйверов

### Драйвер `database`

При использовании драйвера сессии `database` вам необходимо убедиться, что у вас есть таблица базы данных, содержащая данные сеанса. Обычно это включено в стандартный файл Laravel `0001_01_01_000000_create_users_table.php` [МИГРАЦИИ БАЗЫ ДАННЫХ](#); однако, если по какой-либо причине у вас нет таблицы `sessions`, вы можете использовать Artisan-команду `make:session-table` для создания этой миграции:

```
php artisan make:session-table
```

```
php artisan migrate
```

### Redis

Перед использованием Redis с Laravel вам нужно будет либо установить расширение PHP `PhpRedis` через PECL, либо установить пакет `predis/predis` (~ 1.0)

через Composer. Для получения дополнительной информации о настройке Redis обратитесь к [документации Redis Laravel](#).

Переменная среды `SESSION_CONNECTION` или опция `connection` в файле конфигурации `session.php` может использоваться для указания того, какое соединение Redis используется для хранения сеанса.

## # Взаимодействие с сессией

### Получение данных

В Laravel есть два основных способа работы с данными сессии: через глобальный помощник `session` или через экземпляр `Request`. Во-первых, давайте посмотрим на доступ к сессии через экземпляр `Request`, тип которого может быть объявлен в замыкании маршрута или методе контроллера. Помните, что зависимости методов контроллера автоматически внедряются через [контейнер служб](#) Laravel:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать профиль конкретного пользователя.
     */
    public function show(Request $request, string $id): View
    {
        $value = $request->session()->get('key');

        //

        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

```
    }  
}
```

Когда вы извлекаете элемент из сессии, вы также можете передать значение по умолчанию в качестве второго аргумента метода `get`. Это значение по умолчанию будет возвращено, если указанный ключ не существует в сессии. Если вы передаете замыкание в качестве значения по умолчанию методу `get`, а запрошенный ключ не существует, то будет выполнено замыкание с последующим возвратом его результата:

```
$value = $request->session()->get('key', 'default');  
  
$value = $request->session()->get('key', function () {  
    return 'default';  
});
```

## Глобальный помощник `session`

Вы также можете использовать глобальную помощник `session` для получения / сохранения данных сессии. Когда помощник `session` вызывается с одним строковым аргументом, тогда он возвращает значение этого ключа сессии. Когда помощник вызывается с массивом пар ключ / значение, эти значения сохраняются в сессию:

```
Route::get('/home', function () {  
    // Получить часть данных из сессии ...  
    $value = session('key');  
  
    // Получить часть данных из сессии с указанием значения по умолчанию ...  
    $value = session('key', 'default');  
  
    // Сохранить часть данных в сессию ...  
    session(['key' => 'value']);  
});
```

Существует небольшая практическая разница между использованием сессии через экземпляр HTTP-запроса и использованием глобального помощника `session`. Оба метода [тестируемые](#) с

помощью метода `assertSessionHas`, который доступен во всех ваших тестах.

## Получение всех данных сессии

Если вы хотите получить все данные сессии, то вы можете использовать метод `all`:

```
$data = $request->session()->all();
```

## Получение части данных сессии

Методы `only` и `except` могут быть использованы для извлечения подмножества данных сессии:

```
$data = $request->session()->only(['username', 'email']);
```

```
$data = $request->session()->except(['username', 'email']);
```

## Определение наличия элемента в сессии

Чтобы определить, присутствует ли элемент в сессии, вы можете использовать метод `has`. Метод `has` возвращает `true`, если элемент присутствует, и не равен `null`:

```
if ($request->session()->has('users')) {  
    // ...  
}
```

Чтобы определить, присутствует ли элемент в сессии, даже если его значение равно `null`, то вы можете использовать метод `exists`:

```
if ($request->session()->exists('users')) {  
    // ...  
}
```

Чтобы определить, отсутствует ли элемент в сессии, вы можете использовать метод `missing`. Метод `missing` возвращает `true`, если элемент имеет значение `null`

или если элемент отсутствует:

```
if ($request->session()->missing('users')) {  
    // ...  
}
```

## Сохранение данных

Для сохранения данных в сессии вы обычно будете использовать метод `put` экземпляра запроса или глобального помощника `session`:

```
// Через экземпляр запроса ...  
$request->session()->put('key', 'value');  
  
// Через глобальный помощник «session» ...  
session(['key' => 'value']);
```

## Добавление в массив значений сессии

Метод `push` используется для вставки нового значения в значение сессии, которое является массивом. Например, если ключ `user.teams` содержит массив названий команд, то вы можете поместить новое значение в массив следующим образом:

```
$request->session()->push('user.teams', 'developers');
```

## Получение с последующим удалением элемента

Метод `pull` извлекает и удаляет элемент из сессии единым выражением:

```
$value = $request->session()->pull('key', 'default');
```

## Увеличение и уменьшение отдельных значений в сессии

Если данные вашей сессии содержат целое число, которое вы хотите увеличить или уменьшить, то вы можете использовать методы `increment` и `decrement`:

```
$request->session()->increment('count');
```

```
$request->session()->increment('count', $incrementBy = 2);  
  
$request->session()->decrement('count');  
  
$request->session()->decrement('count', $decrementBy = 2);
```

## Кратковременные данные

По желанию можно сохранить элементы в сессии только для следующего запроса. Вы можете сделать это с помощью метода `flash`. Данные, хранящиеся в сессии с использованием этого метода, будут доступны немедленно и во время следующего HTTP-запроса. После следующего HTTP-запроса данные будут удалены. Кратковременные данные в первую очередь полезны для краткосрочных статусных сообщений:

```
$request->session()->flash('status', 'Task was successful!');
```

Если вам нужно сохранить кратковременные данные для нескольких запросов, то вы можете использовать метод `reflash`, который сохранит все данные для дополнительного запроса. Если вам нужно сохранить конкретные кратковременные данные, то вы можете использовать метод `keep`:

```
$request->session()->reflash();  
  
$request->session()->keep(['username', 'email']);
```

Чтобы сохранить ваши кратковременные данные только для текущего запроса, вы можете использовать метод `now`:

```
$request->session()->now('status', 'Task was successful!');
```

## Удаление данных

Метод `forget` удалит часть данных из сессии. Если вы хотите удалить все данные из сессии, то вы можете использовать метод `flush`:

```
// Удалить единственный ключ ...  
$request->session()->forget('name');
```

```
// Удалить несколько ключей ...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

## Пересоздание идентификатора сессии

Пересоздание идентификатора сессии часто выполняется для предотвращения использования злоумышленниками атаки, называемой [фиксацией сессии](#), на ваше приложение.

Laravel автоматически пересоздает идентификатор сессии во время аутентификации, если вы используете один из [стартовых комплектов приложений Laravel](#) или [Laravel Fortify](#); однако, если вам необходимо вручную повторно сгенерировать идентификатор сессии, то вы можете использовать метод `regenerate`:

```
$request->session()->regenerate();
```

Если вам нужно повторно сгенерировать идентификатор сессии и удалить все данные из нее одним выражением, то вы можете использовать метод `invalidate`:

```
$request->session()->invalidate();
```

## # Блокировка сессии

Чтобы использовать блокировку сессии, ваше приложение должно использовать драйвер кеша, поддерживающий [атомарные блокировки](#). В настоящее время этими драйверами кеширования являются `memcached`, `dynamodb`, `redis` и `database`, `file` и `array`. Кроме того, вы не можете использовать драйвер сессии `cookie`.

По умолчанию Laravel позволяет выполнять запросы, использующие одну и ту же сессию, одновременно. Так, например, если вы используете HTTP-библиотеку JavaScript для выполнения двух HTTP-запросов к вашему приложению, то они будут выполняться одновременно. Для многих приложений это не проблема; однако потеря данных сессии может произойти в небольшом подмножестве приложений, выполняющих одновременные запросы к двум различным конечным точкам приложения, которые записывают данные в сессию.

Чтобы смягчить это, Laravel предлагает функциональность, которая позволяет ограничивать количество одновременных запросов для текущей сессии. Для начала вы можете просто привязать метод `block` к определению вашего маршрута. В этом примере входящий запрос к конечной точке `/profile` получит блокировку сессии. Пока эта блокировка удерживается, любые входящие запросы к конечным точкам `/profile` или `/order` с одним и тем же идентификатором сессии будут ждать завершения выполнения первого запроса, прежде чем они будут выполнены:

```
Route::post('/profile', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)
```

Метод `block` принимает два необязательных аргумента. Первый аргумент, принимаемый методом `block` – это максимальное количество секунд, в течение которых блокировка сессии должна удерживаться, прежде чем она будет снята. Конечно, если выполнение запроса завершится до этого времени, блокировка будет снята раньше.

Второй аргумент, принимаемый методом `block` – это количество секунд, в течение которых запрос должен ждать при попытке получить блокировку сессии. Если запрос не сможет получить блокировку сессии в течение указанного количества секунд, то будет выброшено исключение [Illuminate\Contracts\Cache\LockTimeoutException](#).

Если ни один из этих аргументов не передан, то блокировка будет получена максимум на `10` секунд, а запросы будут ждать максимум `10` секунд при попытке получить блокировку:

```
Route::post('/profile', function () {
    // ...
})->block()
```

## # Добавление пользовательских драйверов сессии

### Реализация пользовательского драйвера

Если ни один из существующих драйверов сессии не соответствует потребностям вашего приложения, то Laravel позволяет написать собственный обработчик сессии. Ваш собственный драйвер сессии должен реализовывать [SessionHandlerInterface](#), встроенный в PHP. Этот интерфейс содержит всего несколько простых методов. Заготовка реализации MongoDB выглядит следующим образом:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionId) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

Laravel не содержит каталога для хранения ваших расширений. Вы можете разместить их где угодно. В этом примере мы создали каталог [Extensions](#) для размещения [MongoSessionHandler](#).

Поскольку цель этих методов не совсем понятна, давайте быстро рассмотрим, что делает каждый из этих методов:

- Метод `open` обычно используется в файловых системах хранения сессии. Поскольку Laravel поставляется с драйвером `file` сессии, за редким исключением вам понадобится что-либо вставлять в этот метод. Вы можете просто оставить этот метод пустым.
- Метод `close`, как и метод `open`, также обычно не учитывается. Для большинства драйверов в этом нет необходимости.
- Метод `read` должен возвращать строковую версию данных сессии, связанных с переданным `$sessionId`. Нет необходимости выполнять сериализацию или другое кодирование при получении или хранении данных сессии в вашем драйвере, поскольку Laravel выполнит сериализацию за вас.
- Метод `write` должен записать переданную строку `$data`, связанную с `$sessionId`, в какую-нибудь постоянную систему хранения, такую как MongoDB или другую систему хранения по вашему выбору. Опять же, вам не следует выполнять сериализацию – Laravel сделает это за вас.
- Метод `destroy` должен удалить данные, связанные с `$sessionId` из постоянного хранилища.
- Метод `gc` должен уничтожить все данные сессии, которые старше указанного `$lifetime`, которое является временной меткой UNIX. Для самоуничтожающихся систем, таких как Memcached и Redis, этот метод можно оставить пустым.

## Регистрация пользовательского драйвера

Как только ваш драйвер будет реализован, вы готовы зарегистрировать его в Laravel. Чтобы добавить дополнительные драйверы в серверную часть сессии Laravel, вы можете использовать метод `extend` [фасада Session](#). Вы должны вызвать метод `extend` в методе `boot` [поставщика службы](#). Вы можете сделать это в уже существующем `App\Providers\AppServiceProvider` или создать совершенно новый поставщик:

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
```

```
{  
    /**  
     * Регистрация любых служб приложения.  
     */  
    public function register(): void  
    {  
        // ...  
    }  
  
    /**  
     * Загрузка любых служб приложения.  
     */  
    public function boot(): void  
    {  
        Session::extend('mongo', function (Application $app) {  
            // Return an implementation of SessionHandlerInterface...  
            return new MongoSessionHandler;  
        });  
    }  
}
```

После регистрации драйвера сеанса вы можете указать драйвер `mongo` в качестве драйвера сеанса вашего приложения, используя переменную среды `SESSION_DRIVER` или в файле конфигурации приложения `config/session.php`.

# Валидация

## # Введение

## # Быстрый старт

- # Определение маршрутов
- # Создание контроллера
- # Написание логики валидации
- # Отображение ошибок валидации
- # Повторное заполнение форм
- # Примечание о необязательных полях
- # Формат ответа об ошибках валидации

## # Валидация запроса формы

- # Создание запросов формы
- # Авторизация запросов
- # Корректировка сообщений об ошибках
- # Подготовка входящих данных для валидации

## # Создание валидатора по требованию

- # Автоматическое перенаправление
- # Именованные коллекции ошибок
- # Корректировка сообщений об ошибках
- # Выполнение дополнительной валидации

## # Работа с проверенными данными

## # Работа с сообщениями об ошибках

- # Указание пользовательских сообщений в языковых файлах
- # Указание атрибутов в языковых файлах
- # Указание пользовательских имен для атрибутов в языковых файлах

## # Доступные правила валидации

## # Условное добавление правил

## # Валидация массивов

- # Проверка входных данных вложенного массива
- # Индексы и позиции в сообщениях об ошибках

- # Валидация файлов
- # Валидация паролей
- # Пользовательские правила валидации
  - # Использование класса Rule
  - # Использование замыканий
  - # Неявные правила

## # Введение

Laravel предлагает несколько подходов для проверки входящих данных вашего приложения. Например, метод `validate`, доступен для всех входящих HTTP-запросов. Однако мы обсудим и другие подходы к валидации.

Laravel содержит удобные правила валидации, применяемые к данным, включая валидацию на уникальность значения в конкретной таблице базы данных. Мы подробно рассмотрим каждое из этих правил валидации, чтобы вы были знакомы со всеми особенностями валидации Laravel.

## # Быстрый старт

Чтобы узнать о мощных функциях валидации Laravel, давайте рассмотрим полный пример валидации формы и отображения сообщений об ошибках конечному пользователю. Прочитав этот общий обзор, вы сможете получить представление о том, как проверять данные входящего запроса с помощью Laravel:

## Определение маршрутов

Во-первых, предположим, что в нашем файле `routes/web.php` определены следующие маршруты:

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

Маршрут `GET` отобразит форму для пользователя для создания нового сообщения в блоге, а маршрут `POST` сохранит новое сообщение в базе данных.

## Создание контроллера

Теперь давайте взглянем на простой контроллер, который обрабатывает запросы, входящие на эти маршруты. Пока оставим метод `store` пустым:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\View\View;

class PostController extends Controller
{
    /**
     * Показать форму для создания нового сообщения в блоге.
     */
    public function create(): View
    {
        return view('post.create');
    }

    /**
     * Сохранить новую запись в блоге.
     */
    public function store(Request $request): RedirectResponse
    {
        // Выполнить валидацию и сохранить сообщение ...

        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

## Написание логики валидации

Теперь мы готовы заполнить наш метод `store` логикой для валидации нового сообщения в блоге. Для этого мы будем использовать метод `validate`, предоставляемый объектом `Illuminate\Http\Request`. Если правила валидации будут

пройдены, то ваш код продолжит нормально выполняться; однако, если проверка не пройдена, то будет создано исключение `Illuminate\Validation\ValidationException` и соответствующий ответ об ошибке будет автоматически отправлен обратно пользователю.

Если валидации не пройдена во время традиционного HTTP-запроса, то будет сгенерирован ответ-перенаправление на предыдущий URL-адрес. Если входящий запрос является XHR-запросом, то будет возвращен [JSON-ответ, содержащий сообщения об ошибках валидации](#).

Чтобы лучше понять метод `validate`, давайте вернемся к методу `store`:

```
/*
 * Сохранить новую запись в блоге.
 */
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // Запись блога корректна ...
    return redirect('/posts');
}
```

Как видите, правила валидации передаются в метод `validate`. Не волнуйтесь – все доступные правила валидации [задокументированы](#). Опять же, если проверка не пройдена, то будет автоматически сгенерирован корректный ответ. Если проверка пройдет успешно, то наш контроллер продолжит нормальную работу.

В качестве альтернативы правила валидации могут быть указаны как массивы правил вместо одной строки с разделителями `|`:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Кроме того, вы можете использовать метод `validateWithBag` для валидации запроса и сохранения любых сообщений об ошибках в [именованную коллекцию ошибок](#):

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

## Прекращение валидации при возникновении первой ошибки

По желанию можно прекратить выполнение правил валидации для атрибута после первой ошибки. Для этого присвойте атрибуту правило `bail`:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

В этом примере, если правило `unique` для атрибута `title` не будет пройдено, то правило `max` не будет выполняться. Правила будут проверяться в порядке их назначения.

## Примечание о вложенных атрибутах

Если входящий HTTP-запрос содержит данные «вложенных» полей, то вы можете указать эти поля в своих правилах валидации, используя «точечную нотацию»:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

С другой стороны, если имя вашего поля буквально содержит точку, то вы можете явно запретить ее интерпретацию как часть «точечной нотации», экранировав точку с помощью обратной косой черты:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
```

```
'v1\\.0' => 'required',  
]);
```

## Отображение ошибок валидации

Итак, что, если поля входящего запроса не проходят указанные правила валидации? Как упоминалось ранее, Laravel автоматически перенаправит пользователя обратно в его предыдущее местоположение. Кроме того, все ошибки валидации и [входящие данные запроса](#) будут автоматически [записаны в сессию](#).

Переменная `$errors` используется во всех шаблонах вашего приложения благодаря посреднику `Illuminate\View\Middleware\ShareErrorsFromSession`, который включен в группу посредников `web`. Пока применяется этот посредник, в ваших шаблонах всегда будет доступна переменная `$errors`, что позволяет вам предполагать, что переменная `$errors` всегда определена и может безопасно использоваться.

Переменная `$errors` будет экземпляром `Illuminate\Support\MessageBag`. Для получения дополнительной информации о работе с этим объектом [ознакомьтесь с его документацией](#).

Итак, в нашем примере пользователь будет перенаправлен на метод нашего контроллера `create`, в случае, если валидация завершится неудачно, что позволит нам отобразить сообщения об ошибках в шаблоне:

```
<!-- /resources/views/post/create.blade.php -->  
  
<h1>Создание поста блога</h1>  
  
@if ($errors->any())  
    <div class="alert alert-danger">  
        <ul>  
            @foreach ($errors->all() as $error)  
                <li>{{ $error }}</li>  
            @endforeach  
        </ul>  
    </div>  
@endif  
  
<!-- Форма для создания поста блога -->
```

## Корректировка сообщений об ошибках

Каждое встроенное правило валидации Laravel содержит сообщение об ошибке, которое находится в файле `lang/en/validation.php` вашего приложения. Если ваше приложение не содержит директорию `lang`, вы можете указать Laravel создать ее с помощью команды Artisan `lang:publish`.

В файле `lang/en/validation.php` вы найдете запись о переводе для каждого правила валидации. Вы можете изменять или модифицировать эти сообщения в зависимости от потребностей вашего приложения.

Кроме того, вы можете скопировать этот файл в каталог перевода другого языка, чтобы перевести сообщения на язык вашего приложения. Чтобы узнать больше о локализации Laravel, ознакомьтесь с полной [документацией по локализации](#).

По умолчанию стандартная структура приложения Laravel не включает директорию `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды Artisan `lang:publish`.

## XHR-запросы и валидация

В этом примере мы использовали традиционную форму для отправки данных в приложение. Однако, многие приложения получают запросы XHR с фронтенда с использованием JavaScript. При использовании метода `validate`, во время выполнения XHR-запроса, Laravel не будет генерировать ответ-перенаправление. Вместо этого Laravel генерирует [JSON-ответ, содержащий все ошибки валидации](#). Этот ответ JSON будет отправлен с кодом 422 состояния HTTP.

## Директива @error

Вы можете использовать директиву `@error` Blade, чтобы быстро определить, существуют ли сообщения об ошибках валидации для конкретного атрибута, включая сообщения об ошибках в именованной коллекции ошибок. В директиве `@error` вы можете вывести содержимое переменной `$message` для отображения сообщения об ошибке:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input
    id="title"
    type="text"
    name="title"
    class="@error('title') is-invalid @enderror"
/>

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Если вы используете [именованные коллекции ошибок](#), вы можете передать имя коллекции ошибок в качестве второго аргумента директивы `@error`:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

## Повторное заполнение форм

Когда Laravel генерирует ответ-перенаправление из-за ошибки валидации, фреймворк автоматически [краткосрочно записывает все входные данные запроса в сессию](#). Это сделано для того, чтобы вы могли удобно получить доступ к входным данным во время следующего запроса и повторно заполнить форму, которую пользователь попытался отправить.

Чтобы получить входные данные предыдущего запроса, вызовите метод `old` экземпляра `Illuminate\Http\Request`. Метод `old` извлечет ранее записанные входные данные из [сессии](#):

```
$title = $request->old('title');
```

Laravel также содержит глобального помощника `old`. Если вы показываете входные данные прошлого запроса в [шаблоне Blade](#), то удобнее использовать помощник `old` для повторного заполнения формы. Если для какого-то поля не были предоставлены данные в прошлом запросе, то будет возвращен `null`:

```
<input type="text" name="title" value="{{ old('title') }}>
```

## Примечание о необязательных полях

По умолчанию Laravel содержит посредников `App\Http\Middleware\TrimStrings` и `App\Http\Middleware\ConvertEmptyStringsToNull` в глобальном стеке посредников вашего приложения. Первый из упомянутых посредников будет автоматически обрезать все входящие строковые поля запроса, а второй – конвертировать любые пустые строковые поля в `null`. Из-за этого вам часто нужно будет помечать ваши «необязательные» поля запроса как `nullable`, если вы не хотите, чтобы валидатор не считал такие поля недействительными. Например:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

В этом примере мы указываем, что поле `publish_at` может быть либо `null`, либо допустимым представлением даты. Если модификатор `nullable` не добавлен в определение правила, валидатор сочтет `null` недопустимой датой.

## Формат ответа об ошибках валидации

Когда ваше приложение генерирует исключение `Illuminate\Validation\ValidationException`, и входящий HTTP-запрос ожидает JSON-ответ, Laravel автоматически форматирует сообщения об ошибке и возвращает HTTP-ответ 422 `Unprocessable Entity`.

Ниже приведен пример формата JSON-ответа для ошибок валидации. Обратите внимание, что вложенные ключи ошибок приводятся к формату “точечной” нотации:

```
{
    "message": "The team name must be a string. (and 4 more errors)",
    "errors": {
        "team_name": [
            "The team name must be a string.",
            "The team name must be at least 1 characters."
        ]
    }
}
```

```
        ],
        "authorization.role": [
            "The selected authorization.role is invalid."
        ],
        "users.0.email": [
            "The users.0.email field is required."
        ],
        "users.2.email": [
            "The users.2.email must be a valid email address."
        ]
    }
}
```

## # Валидация запроса формы

### Создание запросов формы

Для более сложных сценариев валидации вы можете создать «запрос формы».

Запрос формы – это ваш класс запроса, который инкапсулирует свою собственную логику валидации и авторизации. Чтобы сгенерировать новый запрос формы, используйте команду `make:request Artisan`:

```
php artisan make:request StorePostRequest
```

Эта команда поместит новый класс запроса формы в каталог `app/Http/Requests` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его, когда вы запустите команду `make:request`. Каждый запрос формы, созданный Laravel, имеет два метода: `authorize` и `rules`.

Как вы могли догадаться, метод `authorize` проверяет, может ли текущий аутентифицированный пользователь выполнить действие, представленное запросом, а метод `rules` возвращает правила валидации, которые должны применяться к данным запроса:

```
/**
 * Получить массив правил валидации, которые будут применены к запросу.
 *
 * @return array<string, \Illuminate\Contracts\Validation\ValidationRule|array<mixed>
 */
public function rules(): array
```

```
{  
    return [  
        'title' => 'required|unique:posts|max:255',  
        'body' => 'required',  
    ];  
}
```

Вы можете объявить любые зависимости, которые вам нужны, в сигнатуре метода `rules`. Они будут автоматически извлечены через [контейнер служб Laravel](#).

Итак, как анализируются правила валидации? Все, что вам нужно сделать, это объявить зависимость от запроса в методе вашего контроллера. Входящий запрос формы проверяется до вызова метода контроллера, что означает, что вам не нужно загромождать контроллер какой-либо логикой валидации:

```
/**  
 * Сохранить новую запись в блоге.  
 */  
public function store(StorePostRequest $request): RedirectResponse  
{  
    // Входящий запрос прошел валидацию...  
  
    // Получить проверенные входные данные...  
    $validated = $request->validated();  
  
    // Получить часть проверенных входных данных...  
    $validated = $request->safe()->only(['name', 'email']);  
    $validated = $request->safe()->except(['name', 'email']);  
  
    // Сохранить запись в блоге...  
  
    return redirect('/posts');  
}
```

При неуспешной валидации будет сгенерирован ответ-перенаправление, чтобы отправить пользователя обратно в его предыдущее местоположение. Ошибки также будут краткосрочно записаны в сессию, чтобы они были доступны

для отображения. Если запрос был XHR-запросом, то пользователю будет возвращен HTTP-ответ с кодом состояния 422, включая [JSON-представление ошибок валидации..](#)

Хотите добавить мгновенную валидацию формы для вашего фронтенда Laravel, построенного с использованием Inertia? Посмотрите [Laravel Precognition](#).

## Выполнение дополнительной валидации

Иногда вам нужно выполнить дополнительную валидацию после завершения начальной валидации. Это можно сделать с использованием метода `after` запроса формы.

Метод `after` должен возвращать массив вызываемых объектов или замыканий, которые будут вызваны после завершения валидации. Предоставленные вызываемые объекты будут получать экземпляр `Illuminate\Validation\Validator`, позволяя вам добавлять дополнительные сообщения об ошибках, если это необходимо:

```
use Illuminate\Validation\Validator;

/**
 * Get the "after" validation callables for the request.
 */
public function after(): array
{
    return [
        function (Validator $validator) {
            if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add(
                    'field',
                    'Something is wrong with this field!'
                );
            }
        ],
    ];
}
```

Как указано, массив, возвращаемый методом `after`, также может содержать вызываемые классы. Метод `__invoke` этих классов получит экземпляр `Illuminate\Validation\Validator`:

```
use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;
use Illuminate\Validation\Validator;

/**
 * Get the "after" validation callables for the request.
 */
public function after(): array
{
    return [
        new ValidateUserStatus,
        new ValidateShippingTime,
        function (Validator $validator) {
            //
        }
    ];
}
```

## Прекращение валидации после первой неуспешной проверки

Добавив свойство `$stopOnFirstFailure` вашему классу запроса, вы можете сообщить валидатору, что он должен прекратить валидацию всех атрибутов после возникновения первой ошибки валидации:

```
/**
 * Остановить валидацию после первой неуспешной проверки.
 *
 * @var bool
 */
protected $stopOnFirstFailure = true;
```

## Настройка ответа-перенаправления

Как обсуждалось ранее, будет сгенерирован ответ-перенаправление, чтобы отправить пользователя обратно в его предыдущее местоположение, когда проверка формы запроса не удалась. Однако вы можете настроить это поведение. Для этого определите свойство `$redirect` в вашем классе запроса:

```
/**  
 * URI, на который следует перенаправлять пользователей в случае сбоя проверки.  
 *  
 * @var string  
 */  
protected $redirect = '/dashboard';
```

Или, если вы хотите перенаправить пользователей на именованный маршрут, вы можете вместо этого определить свойство `$redirectToRoute`:

```
/**  
 * Маршрут, на который следует перенаправлять пользователей в случае сбоя проверки.  
 *  
 * @var string  
 */  
protected $redirectToRoute = 'dashboard';
```

## Авторизация запросов

Класс запроса формы также содержит метод `authorize`. В рамках этого метода вы можете определить, действительно ли аутентифицированный пользователь имеет право изменять текущий ресурс. Например, вы можете определить, действительно ли пользователь владеет комментарием в блоге, который он пытается обновить. Скорее всего, вы будете взаимодействовать с вашими [шлюзами и политиками авторизации](#) в этом методе:

```
use App\Models\Comment;  
  
/**  
 * Определить, уполномочен ли пользователь выполнить этот запрос.  
 */  
public function authorize(): bool  
{  
    $comment = Comment::find($this->route('comment'));  
  
    return $comment && $this->user()->can('update', $comment);  
}
```

Поскольку все запросы формы расширяют базовый класс запросов Laravel, мы можем использовать метод `user` для доступа к текущему аутентифицированному

пользователю. Также обратите внимание на вызов метода `route` в приведенном выше примере. Этот метод обеспечивает вам доступ к параметрам URI, определенным для вызываемого маршрута, таким как параметр `{comment}` в приведенном ниже примере:

```
Route::post('/comment/{comment}');
```

Поэтому, если ваше приложение использует [привязку модели к маршруту](#), ваш код можно сделать еще более кратким, обратившись к разрешенной модели в качестве свойства запроса:

```
return $this->user()->can('update', $this->comment);
```

Если метод `authorize` возвращает `false`, то будет автоматически возвращен HTTP-ответ с кодом состояния 403, и метод вашего контроллера не будет выполнен.

Если вы планируете обрабатывать логику авторизации для запроса в другой части вашего приложения, то вы можете полностью удалить метод `authorize` или просто вернуть `true`:

```
/**  
 * Определить, уполномочен ли пользователь выполнить этот запрос.  
 */  
public function authorize(): bool  
{  
    return true;  
}
```

Вы можете объявить любые зависимости, которые вам нужны, в сигнатуре метода `authorize`. Они будут автоматически извлечены через [контейнер служб Laravel](#).

## Корректировка сообщений об ошибках

Вы можете изменить сообщения об ошибках, используемые в запросе формы, переопределив метод `messages`. Этот метод должен возвращать массив пар атрибут / правило и соответствующие им сообщения об ошибках:

```
/**  
 * Получить сообщения об ошибках для определенных правил валидации.  
 *  
 * @return array<string, string>  
 */  
public function messages(): array  
{  
    return [  
        'title.required' => 'A title is required',  
        'body.required' => 'A message is required',  
    ];  
}
```

## Корректировка атрибутов валидации

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute`. Если вы хотите, чтобы заполнитель `:attribute` вашего сообщения валидации был заменен другим именем атрибута, то вы можете указать собственные имена, переопределив метод `attributes`. Этот метод должен возвращать массив пар атрибут / имя:

```
/**  
 * Получить пользовательские имена атрибутов для формирования ошибок валидатора.  
 *  
 * @return array<string, string>  
 */  
public function attributes(): array  
{  
    return [  
        'email' => 'email address',  
    ];  
}
```

## Подготовка входящих данных для валидации

Если вам необходимо подготовить или обработать какие-либо данные из запроса перед применением правил валидации, то вы можете использовать метод `prepareForValidation`:

```
use Illuminate\Support\Str;

/**
 * Подготовить данные для валидации.
 */
protected function prepareForValidation(): void
{
    $this->merge([
        'slug' => Str::slug($this->slug),
    ]);
}
```

Точно так же, если вам нужно нормализовать какие-либо данные запроса после завершения валидации, вы можете использовать метод `passedValidation`:

```
/**
 * Обработка успешной попытки валидации:
 */
protected function passedValidation(): void
{
    $this->replace(['name' => 'Taylor']);
}
```

## # Создание валидатора по требованию

Если вы не хотите использовать метод `validate` запроса, то вы можете создать экземпляр валидатора вручную, используя [фасад Validator](#). Метод `make` фасада генерирует новый экземпляр валидатора:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Сохранить новую запись в блоге.
     */
}
```

```

public function store(Request $request): RedirectResponse
{
    $validator = Validator::make($request->all(), [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    if ($validator->fails()) {
        return redirect('/post/create')
            ->withErrors($validator)
            ->withInput();
    }

    // Получить проверенные данные...
    $validated = $validator->validated();

    // Получить часть проверенных данных...
    $validated = $validator->safe()->only(['name', 'email']);
    $validated = $validator->safe()->except(['name', 'email']);

    // Сохранить сообщение блога ...

    return redirect('/posts');
}
}

```

Первым аргументом, переданным методу `make`, являются проверяемые данные. Второй аргумент – это массив правил валидации, которые должны применяться к данным.

После определения того, что запрос не прошел валидацию с помощью метода `fails`, вы можете использовать метод `withErrors` для передачи сообщений об ошибках в сессию. При использовании этого метода переменная `$errors` будет автоматически передана вашим шаблонам после перенаправления, что позволит вам легко отобразить их обратно пользователю. Метод `withErrors` принимает экземпляр валидатора, экземпляр `MessageBag` или обычный массив PHP.

## Прекращение валидации после первой неуспешной проверки

Метод `stopOnFirstFailure` проинформирует валидатор о том, что он должен прекратить валидацию всех атрибутов после возникновения первой ошибки валидации:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

## Автоматическое перенаправление

Если вы хотите создать экземпляр валидатора вручную, но по-прежнему воспользоваться преимуществами автоматического перенаправления, предлагаемого методом `validate` HTTP-запроса, вы можете вызвать метод `validate` созданного экземпляра валидатора. Пользователь будет автоматически перенаправлен или, в случае запроса XHR, будет возвращен ответ JSON, если валидация будет не успешной:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Вы можете использовать метод `validateWithBag` для сохранения сообщений об ошибках в именованной коллекции ошибок, если валидация будет не успешной:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validateWithBag('post');
```

## Именованные коллекции ошибок

Если у вас есть несколько форм на одной странице, то вы можете задать имя экземпляру `MessageBag`, содержащий ошибки валидации, что позволит вам получать сообщения об ошибках для конкретной формы. Чтобы добиться этого, передайте имя в качестве второго аргумента в метод `withErrors`:

```
return redirect('/register')->withErrors($validator, 'login');
```

Затем, вы можете получить доступ к именованному экземпляру `MessageBag` из переменной `$errors`:

```
 {{ $errors->login->first('email') }}
```

## Корректировка сообщений об ошибках

При необходимости вы можете предоставить собственные сообщения об ошибках, которые должен использовать экземпляр валидатора вместо сообщений об ошибках по умолчанию, предоставляемых Laravel. Есть несколько способов указать собственные сообщения. Во-первых, вы можете передать собственные сообщения в качестве третьего аргумента методу `Validator::make`:

```
$validator = Validator::make($input, $rules, $messages = [  
    'required' => 'The :attribute field is required.',  
]);
```

В этом примере заполнитель `:attribute` будет заменен фактическим именем проверяемого поля. Вы также можете использовать другие заполнители в сообщениях валидатора. Например:

```
$messages = [  
    'same' => 'The :attribute and :other must match.',  
    'size' => 'The :attribute must be exactly :size.',  
    'between' => 'The :attribute value :input is not between :min - :max.',  
    'in' => 'The :attribute must be one of the following types: :values',  
];
```

## Указание пользовательского сообщения для конкретного атрибута

По желанию можно указать собственное сообщение об ошибке только для определенного атрибута. Вы можете сделать это, используя «точечную нотацию». Сначала укажите имя атрибута, а затем правило:

```
$messages = [  
    'email.required' => 'We need to know your email address!',  
];
```

## Указание пользовательских имен для атрибутов

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute`, который заменяется именем проверяемого поля или атрибута. Чтобы указать собственные значения, используемые для замены этих заполнителей для конкретных полей, вы можете передать массив ваших атрибутов в качестве четвертого аргумента методу `Validator::make`:

```
$validator = Validator::make($input, $rules, $messages, [
    'email' => 'email address',
]);
```

## Выполнение дополнительной валидации

Иногда вам нужно выполнить дополнительную валидацию после завершения начальной валидации. Это можно сделать с использованием метода `after` валидатора. Метод `after` принимает замыкание или массив вызываемых объектов, которые будут вызваны после завершения валидации. Предоставленные вызываемые объекты будут получать экземпляр `Illuminate\Validation\Validator`, позволяя вам добавлять дополнительные сообщения об ошибках, если это необходимо:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make(/* ... */);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add(
            'field', 'Something is wrong with this field!'
        );
    }
});

if ($validator->fails()) {
    // ...
}
```

Как указано, метод `after` также принимает массив вызываемых объектов, что особенно удобно, если ваша логика “после валидации” инкапсулирована в вызываемых классах, которые будут получать экземпляр `Illuminate\Validation\Validator` через свой метод `__invoke`:

```
use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;

$validator->after([
    new ValidateUserStatus,
    new ValidateShippingTime,
    function ($validator) {
        // ...
    },
]);

```

## # Работа с проверенными данными

После проверки данных входящего запроса с помощью запроса формы или вручную созданного экземпляра валидатора вы можете получить данные входящего запроса, которые действительно прошли проверку. Этого можно добиться несколькими способами. Во-первых, вы можете вызывать метод `validated` в запросе формы или экземпляре валидатора. Этот метод возвращает массив данных, которые были проверены:

```
$validated = $request->validated();

$validated = $validator->validated();
```

В качестве альтернативы вы можете вызвать метод `safe` в запросе формы или экземпляре валидатора. Этот метод возвращает экземпляр `Illuminate\Support\ValidatedInput`. Этот объект предоставляет методы `only`, `except`, и `all` для получения подмножества проверенных данных или всего массива проверенных данных:

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

Кроме того, экземпляр `Illuminate\Support\ValidatedInput` может быть проитерирован и доступен как массив:

```
// Validated data may be iterated...
foreach ($request->safe() as $key => $value) {
    // ...
}

// Validated data may be accessed as an array...
$validated = $request->safe();

$email = $validated['email'];
```

Если вы хотите добавить дополнительные поля к проверенным данным, вы можете вызвать метод `merge`:

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

Если вы хотите получить проверенные данные в виде экземпляра `collection` вы можете вызвать метод `collect`:

```
$collection = $request->safe()->collect();
```

## # Работа с сообщениями об ошибках

После вызова метода `errors` экземпляр `Validator`, вы получите экземпляр `Illuminate\Support\MessageBag`, который имеет множество удобных методов для работы с сообщениями об ошибках. Переменная `$errors`, которая автоматически становится доступной для всех шаблонов, также является экземпляром класса `MessageBag`.

### Получение первого сообщения об ошибке для поля

Чтобы получить первое сообщение об ошибке для указанного поля, используйте метод `first`:

```
$errors = $validator->errors();

echo $errors->first('email');
```

## Получение всех сообщений об ошибках для поля

Если вам нужно получить массив всех сообщений для указанного поля, используйте метод `get`:

```
foreach ($errors->get('email') as $message) {  
    // ...  
}
```

Если вы проверяете массив полей формы, то вы можете получить все сообщения для каждого из элементов массива, используя символ `*`:

```
foreach ($errors->get('attachments.*') as $message) {  
    // ...  
}
```

## Получение всех сообщений об ошибках для всех полей

Чтобы получить массив всех сообщений для всех полей, используйте метод `all`:

```
foreach ($errors->all() as $message) {  
    // ...  
}
```

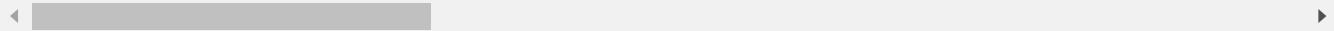
## Определение наличия сообщений для поля

Метод `has` используется для определения наличия сообщений об ошибках для указанного поля:

```
if ($errors->has('email')) {  
    // ...  
}
```

## Указание пользовательских сообщений в языковых файлах

Каждое встроенное правило валидации Laravel содержит сообщение об ошибке, которое на:



В файле `lang/en/validation.php` вы найдете запись о переводе для каждого правила валидации. Вы можете изменять или модифицировать эти сообщения в зависимости от потребностей вашего приложения.

Кроме того, вы можете скопировать этот файл в каталог перевода другого языка, чтобы перевести сообщения на язык вашего приложения. Чтобы узнать больше о локализации Laravel, ознакомьтесь с полной [документацией по локализации](#).

По умолчанию стандартная структура приложения Laravel не включает директорию `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды Artisan `lang:publish`.

## Указание пользовательского сообщения для конкретного атрибута

Вы можете изменить сообщения об ошибках, используемые для указанных комбинаций атрибутов и правил в языковых файлах валидации вашего приложения. Для этого добавьте собственные сообщения в массив `custom` языкового файла `resources/lang/xx/validation.php` вашего приложения:

```
'custom' => [
    'email' => [
        'required' => 'We need to know your email address!',
        'max' => 'Your email address is too long!'
    ],
],
```

## Указание атрибутов в языковых файлах

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute`, который заменяется именем проверяемого поля или

атрибута. Если вы хотите, чтобы часть `:attribute` вашего сообщения валидации была заменена собственным значением, то вы можете указать имя настраиваемого атрибута в массиве `attributes` вашего языкового файла `resources/lang/xx/validation.php`:

```
'attributes' => [
    'email' => 'email address',
],
```

## Указание пользовательских имен для атрибутов в языковых файлах

Некоторые сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:value`, который заменяется текущим значением атрибута запроса. Однако, иногда вам может понадобиться заменить часть `:value` вашего сообщения валидации на собственное значение. Например, рассмотрим следующее правило, которое указывает, что номер кредитной карты требуется обязательно, если для параметра `payment_type` установлено значение `cc`:

```
Validator::make($request->all(), [
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

Если это правило валидации не будет пройдено, то будет выдано следующее сообщение об ошибке:

```
The credit card number field is required when payment type is cc.
```

Вместо того чтобы отображать `cc` в качестве значения типа платежа, вы можете указать более удобное для пользователя представление значения в вашем языковом файле `lang/xx/validation.php`, определив массив `values`:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

По умолчанию стандартная структура приложения Laravel не включает директорию `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды Artisan `lang:publish`.

После определения этого значения правило валидации выдаст следующее сообщение об ошибке:

```
The credit card number field is required when payment type is credit card.
```

## # Доступные правила валидации

Ниже приведен список всех доступных правил валидации и их функций:

[Accepted](#)

[Accepted If](#)

[Active URL](#)

[After \(Date\)](#)

[After Or Equal \(Date\)](#)

[Alpha](#)

[Alpha Dash](#)

[Alpha Numeric](#)

[Array](#)

[Ascii](#)

[Bail](#)

[Before \(Date\)](#)

[Before Or Equal \(Date\)](#)

[Between](#)

[Boolean](#)

[Confirmed](#)

[Contains](#)

[Current Password](#)

[Date](#)  
[Date Equals](#)  
[Date Format](#)  
[Decimal](#)  
[Declined](#)  
[Declined If](#)  
[Different](#)  
[Digits](#)  
[Digits Between](#)  
[Dimensions \(Image Files\)](#)  
[Distinct](#)  
[Doesnt Start With](#)  
[Doesnt End With](#)  
[Email](#)  
[Ends With](#)  
[Enum](#)  
[Exclude](#)  
[Exclude If](#)  
[Exclude Unless](#)  
[Exclude With](#)  
[Exclude Without](#)  
[Exists \(Database\)](#)  
[Extensions](#)  
[File](#)  
[Filled](#)  
[Greater Than](#)  
[Greater Than Or Equal](#)  
[Hex Color](#)  
[Image \(File\)](#)  
[In](#)  
[In Array](#)  
[Integer](#)  
[IP Address](#)  
[JSON](#)  
[Less Than](#)  
[Less Than Or Equal](#)

[List](#)

[Lowercase](#)

[MAC Address](#)

[Max](#)

[MIME Types](#)

[MIME Type By File Extension](#)

[Min](#)

[Min Digits](#)

[Missing](#)

[Missing If](#)

[Missing Unless](#)

[Missing With](#)

[Missing With All](#)

[Multiple Of](#)

[Not In](#)

[Not Regex](#)

[Nullable](#)

[Numeric](#)

[Present](#)

[Present If](#)

[Present Unless](#)

[Present With](#)

[Present With All](#)

[Prohibited](#)

[Prohibited If](#)

[Required If Accepted](#)

[Required If Declined](#)

[Prohibited Unless](#)

[Prohibits](#)

[Regex \(regular expression\)](#)

[Required](#)

[Required If](#)

[Required Unless](#)

[Required With](#)

[Required With All](#)

[Required Without](#)

[Required Without All](#)

[Required Array Keys](#)

[Same](#)

[Size](#)

[Sometimes](#)

[Sometimes](#)

[Starts With](#)

[String](#)

[Timezone](#)

[Unique \(Database\)](#)

[Uppercase](#)

[URL](#)

[ULID](#)

[UUID](#)

## accepted

Проверяемое поле должно иметь значение "yes", "on", 1, "1", true или "true". Применяется для валидации принятия «Условий использования» или аналогичных полей.

## accepted\_if:anotherfield,value,...

Проверяемое поле должно иметь значение "yes", "on", 1, "1", true или "true", если другое проверяемое поле равно указанному значению. Это полезно для валидации принятия «Условий использования» или аналогичных полей.

## active\_url

Проверяемое поле должно иметь допустимую запись А или AAAA в соответствии с функцией dns\_get\_record PHP. Имя хоста указанного URL извлекается с помощью PHP-функции parse\_url перед передачей в dns\_get\_record.

## after:date

Проверяемое поле должно иметь значение после указанной даты. Даты будут переданы в функцию strtotime PHP для преобразования в действительный экземпляр DateTime:

```
'start_date' => 'required|date|after:tomorrow'
```

Вместо передачи строки даты, которая будет проанализирована с помощью [strtotime](#), вы можете указать другое поле для сравнения с датой:

```
'finish_date' => 'required|date|after:start_date'
```

## after\_or\_equal:date

Проверяемое поле должно иметь значение после указанной даты или равное ей. Для получения дополнительной информации см. правило [after](#).

## alpha

Поле, подлежащее валидации, должно состоять исключительно из символов Unicode, входящих в [\p{L}](#) и [\p{M}](#).

Чтобы ограничить это правило валидации символами в диапазоне ASCII ([a-z](#) и [A-Z](#)), вы можете использовать опцию [ascii](#):

```
'username' => 'alpha:ascii',
```

## alpha\_dash

Поле, подлежащее валидации, должно состоять исключительно из символов Unicode, алфавитно-цифровых, входящих в [\p{L}](#), [\p{M}](#), [\p{N}](#), а также из символов ASCII [\( - \)](#) и [\( \\_ \)](#).

Чтобы ограничить это правило валидации символами в диапазоне ASCII ([a-z](#) и [A-Z](#)), вы можете использовать опцию [ascii](#):

```
'username' => 'alpha_dash:ascii',
```

## alpha\_num

Поле, подлежащее валидации, должно состоять исключительно из символов Unicode, алфавитно-цифровых, входящих в `\p{L}`, `\p{M}` и `\p{N}`.

Чтобы ограничить это правило валидации символами в диапазоне ASCII (`a-z` и `A-Z`), вы можете использовать опцию `ascii`:

```
'username' => 'alpha_num:ascii',
```

## array

Проверяемое поле должно быть массивом PHP.

Когда для правила `array` предоставляются дополнительные значения, каждый ключ во входном массиве должен присутствовать в списке значений, предоставленных правилу. В следующем примере ключ `admin` во входном массиве недействителен, поскольку он не содержится в списке значений, предоставленных правилу `array`:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:name,username',
]);
```

В общем, вы всегда должны указывать ключи массива, которые могут присутствовать в вашем массиве.

## ascii

Поле, подлежащее валидации, должно состоять исключительно из 7-битных ASCII-символов.

## bail

Остановить дальнейшее применение правил валидации атрибута после первой неуспешной проверки.

В отличие от правила `bail`, которое прекращает дальнейшую валидацию только конкретного поля, метод `stopOnFirstFailure` сообщит валидатору, что он должен прекратить дальнейшую валидацию всех атрибутов при возникновении первой ошибке:

```
if ($validator->stopOnFirstFailure()->fails()) {  
    // ...  
}
```

## before:date

Проверяемое поле должно быть значением, предшествующим указанной дате. Даты будут переданы в функцию PHP `strtotime` для преобразования в действительный экземпляр `DateTime`. Кроме того, как и в правиле `after`, имя другого проверяемого поля может быть указано в качестве значения `date`.

## before\_or\_equal:date

Проверяемое поле должно иметь значение, предшествующее указанной дате или равное ей. Даты будут переданы в функцию PHP `strtotime` для преобразования в действительный экземпляр `DateTime`. Кроме того, как и в правиле `after`, имя другого проверяемого поля может быть указано в качестве значения `date`.

## between:min,max

Проверяемое поле должно иметь размер между указанными `min` и `max` (включительно). Строки, числа, массивы и файлы оцениваются так же, как и в правиле `size`.

## boolean

Проверяемое поле должно иметь возможность преобразования в логическое значение. Допустимые значения: `true`, `false`, `1`, `0`, `"1"`, и `"0"`.

## confirmed

Проверяемое поле должно иметь совпадающее поле `{field}_confirmation`.

Например, если проверяемое поле – `password`, то поле `password_confirmation` также должно присутствовать во входящих данных.

Вы также можете передать собственное имя поля подтверждения. Например, `confirmed:repeat_username` будет ожидать, что поле `repeat_username` будет соответствовать проверяемому полю.

## contains:foo,bar,...

Проверяемое поле должно представлять собой массив, содержащий все заданные значения параметров.

## current\_password

Проверяемое поле должно соответствовать паролю аутентифицированного пользователя. Вы можете указать [охранника аутентификации](#) используя первый параметр правила:

```
'password' => 'current_password:api'
```

## date

Проверяемое поле должно быть действительной, не относительной датой в соответствии с функцией `strtotime` PHP.

## date\_equals:date

Проверяемое поле должно быть равно указанной дате. Даты будут переданы в функцию `strtotime` PHP для преобразования в действительный экземпляр `DateTime`.

## date\_format:format,...

Проверяемое поле должно соответствовать одному из предоставленных *formats*. При валидации поля следует использовать **либо** `date`, **либо** `date_format`, а не то и другое вместе. Это правило валидации поддерживает все форматы, поддерживаемые классом `DateTime` PHP.

## decimal:min,max

Поле, подлежащее валидации, должно быть числовым и содержать указанное количество десятичных знаков:

```
// Должно иметь ровно два десятичных знака (9.99)...
'price' => 'decimal:2'

// Должно иметь от 2 до 4 десятичных знаков...
'price' => 'decimal:2,4'
```

## declined

Проверяемое поле должно иметь значение "no", "off", 0, "0", false или "false".

## declined\_if:anotherfield,value,...

Проверяемое поле должно иметь значение "no", "off", 0, "0", false или "false", если другое проверяемое поле равно указанному значению.

## different:field

Проверяемое поле должно иметь значение, отличное от *field*.

## digits:value

Целое число, подлежащее валидации, должно иметь точную длину *value*.

## digits\_between:min,max

Целое число, подлежащее валидации, должно иметь длину между переданными *min* и *max*.

## dimensions

Проверяемый файл должен быть изображением, отвечающим ограничениям размеров, указанным в параметрах правила:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Доступные ограничения: *min\_width*, *max\_width*, *min\_height*, *max\_height*, *width*, *height*, *ratio*.

Ограничение *ratio* должно быть представлено как ширина, разделенная на высоту. Это может быть указано дробью вроде `3/2` или числом с плавающей запятой, например `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Поскольку это правило требует нескольких аргументов, вы можете использовать метод `Rule::dimensions` для гибкости составления правила:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxLength(500)->ratio(3 / 2),
    ],
]);
```

## distinct

При валидации массивов проверяемое поле не должно иметь повторяющихся значений:

```
'foo.*.id' => 'distinct'
```

По умолчанию правило `distinct` использует гибкое сравнение переменных. Чтобы использовать жесткое сравнение, вы можете добавить параметр `strict` в определение правила валидации:

```
'foo.*.id' => 'distinct:strict'
```

Вы можете добавить `ignore_case` к аргументам правила валидации, чтобы правило игнорировало различия в использовании регистра букв:

```
'foo.*.id' => 'distinct:ignore_case'
```

## doesn't\_start\_with:foo,bar,...

Поле, подлежащее валидации, не должно начинаться с одного из предоставленных значений.

## doesn't\_end\_with:foo,bar,...

Поле, подлежащее валидации, не должно заканчиваться одним из предоставленных значений.

## email

Проверяемое поле должно быть отформатировано как адрес электронной почты. Это правило валидации использует пакет [egulias/email-validator](#) для проверки адреса электронной почты. По умолчанию применяется валидатор [RFCValidation](#), но вы также можете применить другие стили валидации:

```
'email' => 'email:rfc,dns'
```

В приведенном выше примере будут применяться проверки [RFCValidation](#) и [DNSCheckValidation](#). Вот полный список стилей проверки, которые вы можете применить:

- [rfc: RFCValidation](#)
- [strict: NoRFCWarningsValidation](#)
- [dns: DNSCheckValidation](#)
- [spoof: SpoofCheckValidation](#)
- [filter: FilterEmailValidation](#)
- [filter\\_unicode: FilterEmailValidation::unicode\(\)](#)

Валидатор [filter](#), который использует функцию [filter\\_var](#) PHP, поставляется с Laravel и применялся по умолчанию до Laravel версии 5.8.

Валидаторы `dns` и `spoof` требуют расширения `intl` PHP.

## ends\_with:foo,bar,...

Проверяемое поле должно заканчиваться одним из указанных значений.

## enum

Правило `Enum` это правило на основе класса, которое проверяет, содержит ли проверяемое поле допустимое значение перечисления. Правило `Enum` принимает имя перечисления как единственный аргумент конструктора. При валидации примитивных значений правилу `Enum` следует предоставить поддерживаемое перечисление :

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rule;

$request->validate([
    'status' => [Rule::enum(ServerStatus::class)],
]);
```

Методы `only` и `except` правила `Enum` могут использоваться для ограничения того, какие значения перечисления должны считаться допустимыми:

```
Rule::enum(ServerStatus::class)
->only([ServerStatus::Pending, ServerStatus::Active]);

Rule::enum(ServerStatus::class)
->except([ServerStatus::Pending, ServerStatus::Active]);
```

Метод `when` может использоваться для условного изменения правила `Enum`:

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Rule;

Rule::enum(ServerStatus::class)
->when(
```

```
Auth::user()->isAdmin(),
fn ($rule) => $rule->only(...),
fn ($rule) => $rule->only(...),
);
```

## exclude

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`.

## exclude\_if:anotherfield,value

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле `anotherfield` равно `value`.

Если требуется сложная логика условного исключения, можно воспользоваться методом `Rule::excludeIf`. Этот метод принимает логическое значение или замыкание. При использовании замыкания оно должно возвращать `true` или `false` для указания, следует ли исключить поле, подлежащее валидации:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf(fn () => $request->user()->is_admin),
]);
```

## exclude\_unless:anotherfield,value

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле `anotherfield` не равно `value`. При `value` равном `null` (`exclude_unless: name, null`) проверяемое поле будет исключено, если поле сравнения либо не равно `null`, либо отсутствует в данных запроса.

## exclude\_with:anotherfield

Поле, подлежащее валидации, будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле `anotherfield` присутствует.

## `exclude_without:anotherfield`

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated` если поле `anotherfield` отсутствует.

## `exists:table,column`

Проверяемое поле должно существовать в указанной таблице базы данных.

## Основы использования правила `Exists`

```
'state' => 'exists:states'
```

Если параметр `column` не указан, будет использоваться имя поля. Таким образом, в этом случае правило будет проверять, что таблица базы данных `states` содержит запись со значением столбца `state`, равным значению атрибута запроса `state`.

## Указание пользовательского имени столбца

Вы можете явно указать имя столбца базы данных, которое должно использоваться правилом валидации, поместив его после имени таблицы базы данных:

```
'state' => 'exists:states,abbreviation'
```

Иногда требуется указать конкретное соединение с базой данных, которое будет использоваться для запроса `exists`. Вы можете сделать это, добавив имя подключения к имени таблицы:

```
'email' => 'exists:connection.staff,email'
```

Вместо того чтобы указывать имя таблицы напрямую, вы можете указать модель Eloquent, которая должна использоваться для определения имени таблицы:

```
'user_id' => 'exists:App\Models\User,id'
```

Если вы хотите использовать свой запрос, выполняемый правилом валидации, то вы можете использовать класс `Rule` для гибкости определения правила. В этом примере мы также укажем правила валидации в виде массива вместо использования символа `|` для их разделения:

```
use Illuminate\Database\Query\Builder;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function (Builder $query) {
            return $query->where('account_id', 1);
        }),
    ],
]);
```

Вы можете явно указать имя столбца базы данных, которое должно использоваться правилом `exists`, созданным методом `Rule::exists`, предоставив имя столбца в качестве второго аргумента методу `exists`:

```
'state' => Rule::exists('states', 'abbreviation'),
```

## extensions:foo,bar,...

Проверяемый файл должен иметь назначенное пользователем расширение, соответствующее одному из перечисленных расширений:

```
'photo' => ['required', 'extensions:jpg,png'],
```

Никогда не следует полагаться на валидацию файла только по его назначенному пользователем расширению. Это правило обычно всегда следует

использовать в сочетании с правилами [mimes](#)  
или [mimetypes](#).

## file

Проверяемое поле должно быть успешно загруженным на сервер файлом.

## filled

Проверяемое поле не должно быть пустым, если оно присутствует.

## gt:field

Проверяемое поле должно быть больше указанного *field* или *value*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле [size](#).

## gte:field

Проверяемое поле должно быть больше или равно указанному *field* или *value*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле [size](#).

## hex\_color

Поле, подлежащее валидации, должно содержать допустимое значение цвета в [формате шестнадцатеричного кода](#).

## image

Проверяемый файл должен быть изображением (jpg, jpeg, png, bmp, gif, svg или webp).

## in:foo,bar,...

Проверяемое поле должно быть включено в указанный список значений.

Поскольку это правило часто требует, чтобы вы «[объединяли](#)» массив, то метод [Rule::in](#) можно использовать для гибкого построения правила:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

Когда правило `in` комбинируется с правилом `array` каждое значение во входном массиве должно присутствовать в списке значений, предоставленных правилу `in`. В следующем примере код аэропорта `LAS` во входном массиве недействителен, так как он не содержится в списке аэропортов, предоставленном правилу `in`:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$input = [
    'airports' => ['NYC', 'LAS'],
];

Validator::make($input, [
    'airports' => [
        'required',
        'array',
    ],
    'airports.*' => Rule::in(['NYC', 'LIT']),
]);
```

## in\_array:anotherfield.\*

Проверяемое поле должно существовать в значениях `anotherfield`.

## integer

Проверяемое поле должно быть целым числом.

Это правило валидации не проверяет, что значение поля относится к типу переменной `integer`, а только что значение поля относится к типу, принятому

правилом `FILTER_VALIDATE_INT` PHP. Если вам нужно проверить значение поля в качестве числа, используйте это правило в сочетании с [правилом валидации `numeric`](#).

## ip

Проверяемое поле должно быть IP-адресом.

## ipv4

Проверяемое поле должно быть адресом IPv4.

## ipv6

Проверяемое поле должно быть адресом IPv6.

## json

Проверяемое поле должно быть допустимой строкой JSON.

## lt:field

Проверяемое поле должно быть меньше переданного *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле [size](#).

## lte:field

Проверяемое поле должно быть меньше или равно переданному *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле [size](#).

## lowercase

Поле, подлежащее валидации, должно быть в нижнем регистре.

## list

The field under validation must be an array that is a list. An array is considered a list if its keys consist of consecutive numbers from 0 to `count($array) - 1`. Проверяемое поле должно быть массивом, представляющим собой список. Массив считается списком, если его ключи состоят из последовательных чисел от 0 до `count($array) - 1`.

## mac\_address

Проверяемое поле должно быть MAC-адресом.

## max:value

Проверяемое поле должно быть меньше или равно максимальному *value*. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле [size](#).

## max\_digits:value

Целое число, подлежащее валидации, должно иметь максимальную длину *value*.

## mimetypes:text/plain,...

Проверяемый файл должен соответствовать одному из указанных MIME-типов:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

Чтобы определить MIME-тип загруженного файла, содержимое файла будет прочитано, и фреймворк попытается угадать MIME-тип, который может отличаться от типа, предоставленного клиентом.

## mimes:foo,bar,...

Проверяемый файл должен иметь MIME-тип, соответствующий одному из перечисленных расширений.

```
'photo' => 'mimes:jpg,bmp,png'
```

Несмотря на то, что вам нужно только указать расширения, это правило фактически проверяет MIME-тип файла, читая содержимое файла и угадывая его MIME-тип. Полный список типов MIME и соответствующих им расширений можно найти по следующему адресу:

<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

## MIME-типы и расширения

Это правило валидации не проверяет соответствие между MIME-типом и расширением, которое пользователь назначил файлу. Например, правило валидации `mimes:png` считает файл с допустимым содержимым PNG допустимым изображением PNG, даже если файл назван `photo.txt`. Если вы хотите проверить пользовательское расширение файла, вы можете использовать правило `extensions`.

### `min:value`

Проверяемое поле должно иметь минимальное значение `value`. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

### `min_digits:value`

Целое число, подлежащее валидации, должно иметь минимальную длину `value`.

### `multiple_of:value`

Проверяемое поле должно быть кратным `value`.

### `missing`

Поле, подлежащее валидации, не должно присутствовать во входных данных.

### `missing_if:anotherfield,value,...`

Поле, подлежащее валидации, не должно присутствовать, если поле `anotherfield` равно любому из `value`.

### `missing_unless:anotherfield,value`

Поле, подлежащее валидации, не должно присутствовать, если поле *anotherfield* равно любому из *value*.

## missing\_with:foo,bar,...

Поле, подлежащее валидации, не должно присутствовать *только если* присутствует хотя бы одно из указанных полей.

## missing\_with\_all:foo,bar,...

Поле, подлежащее валидации, не должно присутствовать *только если* присутствуют все указанные поля.

## not\_in:foo,bar,...

Проверяемое поле не должно быть включено в переданный список значений.

Метод `Rule::notIn` используется для гибкого построения правила:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

## not\_regex:pattern

Проверяемое поле не должно соответствовать переданному регулярному выражению.

Под капотом это правило использует функцию `preg_match`, поэтому указанный шаблон должен подчиняться требованиям, предъявляемым к ней, а также включать допустимые разделители. Например: `'email' => 'not_regex:/^.+$/i'`.

При использовании шаблонов `regex` / `not_regex` может потребоваться указать ваши правила валидации с использованием массива вместо использования

разделителей |, особенно если регулярное выражение содержит символ |.

## nullable

Проверяемое поле может быть [null](#).

## numeric

Проверяемое поле должно быть [числовым](#).

## present

Поле, подлежащее валидации, должно существовать во входных данных.

## present\_if:anotherfield,value,...

Поле, подлежащее валидации, должно присутствовать, если поле *anotherfield* равно любому из *value*.

## present\_unless:anotherfield,value

Проверяемое поле должно присутствовать, если поле *anotherfield* не равно какому-либо *value*.

## present\_with:foo,bar,...

Поле, подлежащее валидации, должно присутствовать *только если* присутствует хотя бы одно из указанных полей.

## present\_with\_all:foo,bar,...

Поле, подлежащее валидации, должно присутствовать *только если* присутствуют все указанные поля.

## prohibited

Поле, подлежащее валидации, должно отсутствовать или быть пустым. Поле считается "пустым", если оно соответствует хотя бы одному из

следующих критериев:

- Значение равно `null`.
- Значение является пустой строкой.
- Значение является пустым массивом или пустым объектом, реализующим интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

## `prohibited_if:anotherfield,value,...`

Проверяемое поле должно быть пустым или отсутствовать, если поле `anotherfield` равно любому из указанных `value`. Поле считается “пустым”, если оно соответствует одному из следующих критериев:

- Значение равно `null`.
- Значение является пустой строкой.
- Значение является пустым массивом или пустым объектом, реализующим интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

Если требуется сложная логика условного запрета, можно воспользоваться методом `Rule::prohibitedIf`. Этот метод принимает булево значение или замыкание. Если передано замыкание, оно должно возвращать `true` или `false`, указывая, следует ли запретить поле подлежащее проверке:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf(fn () => $request->user()->is_admin),
]);
```

## `prohibited_unless:anotherfield,value,...`

Проверяемое поле должно быть пустым или отсутствовать, если поле *anotherfield* не равно ни одному из значений *value*. Поле считается “пустым”, если оно соответствует одному из следующих критериев:

- Значение равно `null`.
- Значение является пустой строкой.
- Значение является пустым массивом или пустым объектом, реализующим интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

## `prohibits:anotherfield,...`

Если проверяемое поле не отсутствует и не пусто, все поля в *anotherfield* отсутствовать или быть пустыми. Поле считается “пустым”, если оно соответствует одному из следующих критериев:

- Значение равно `null`.
- Значение является пустой строкой.
- Значение является пустым массивом или пустым объектом, реализующим интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

## `regex:pattern`

Проверяемое поле должно соответствовать переданному регулярному выражению.

Под капотом это правило использует функцию `preg_match`, поэтому указанный шаблон должен подчиняться требованиям, предъявляемым к ней, а также включать допустимые разделители. Например: `'email' => 'regex:/^.+@.+\$/i'`.

При использовании шаблонов `regex` / `not_regex` Может потребоваться указать ваши правила валидации с использованием массива вместо использования

разделителей |, особенно если регулярное выражение содержит символ |.

## required

Проверяемое поле должно присутствовать во входных данных и не быть пустым. Поле считается "пустым", если оно соответствует одному из следующих критериев:

- Значение равно `null`.
- Значение является пустой строкой.
- Значение является пустым массивом или пустым объектом, реализующим интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

## required\_if:anotherfield,value,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* равно любому *value*.

Если вы хотите создать более сложное условие для правила `required_if`, вы можете использовать метод `Rule::requiredIf`. Этот метод принимает логическое значение или замыкание. При выполнении замыкания оно должно возвращать `true` или `false`, чтобы указать, обязательно ли проверяемое поле:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(fn () => $request->user()->is_admin),
]);
```

## required\_if\_accepted:anotherfield,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* равно "yes", "on", 1, "1", true или "true".

## required\_if\_declined:anotherfield,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* равно "no", "off", 0, "0", false или "false".

## required\_unless:anotherfield,value,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* не равно какому-либо *value*. Это также означает, что в данных запроса должно присутствовать *anotherfield*, если *value* не имеет значения null.

Если *value* равно null (`required_unless: name, null`), проверяемое поле будет обязательным, если поле сравнения не равно null или поле сравнения отсутствует в данных запроса.

## required\_with:foo,bar,...

Проверяемое поле должно присутствовать и не быть пустым, *только если* любое из других указанных полей присутствует и не является пустым.

## required\_with\_all:foo,bar,...

Проверяемое поле должно присутствовать и не быть пустым, *только если* все другие указанные поля присутствуют и не являются пустыми.

## required\_without:foo,bar,...

Проверяемое поле должно присутствовать и не быть пустым, *только когда* любое из других указанных полей является пустым или отсутствует.

## required\_without\_all:foo,bar,...

Проверяемое поле должно присутствовать и не быть пустым, *только когда* все другие указанные поля являются пустыми или отсутствуют.

## required\_array\_keys:foo,bar,...

Поле, подлежащее проверке, должно быть массивом и должно содержать как минимум указанные ключи.

## same:field

Переданное *field* должно соответствовать проверяемому полю.

## size:value

Проверяемое поле должно иметь размер, соответствующий переданному *value*. Для строковых данных *value* соответствует количеству символов. Для числовых данных *value* соответствует переданному целочисленному значению (атрибут также должен иметь правило `numeric` или `integer`). Для массива *size* соответствует `count` массива. Для файлов *size* соответствует размеру файла в килобайтах.

Давайте посмотрим на несколько примеров:

```
// Проверяем, что строка содержит ровно 12 символов ...
'title' => 'size:12';

// Проверяем, что передано целое число, равно 10 ...
'seats' => 'integer|size:10';

// Проверяем, что в массиве ровно 5 элементов ...
'tags' => 'array|size:5';

// Проверяем, что размер загружаемого файла составляет ровно 512 килобайт ...
'image' => 'file|size:512';
```

## starts\_with:foo,bar,...

Проверяемое поле должно начинаться с одного из указанных значений.

## string

Проверяемое поле должно быть строкой. Если вы хотите, чтобы поле также могло быть `null`, вы должны назначить этому полю правило `nullable`.

## timezone

Проверяемое поле должно быть допустимым идентификатором часового пояса в соответствии с методом `DateTimeZone::listIdentifiers`.

Аргументы, [принимаемые методом `DateTimeZone::listIdentifiers`](#), также могут быть предоставлены для использования с данным правилом проверки:

```
'timezone' => 'required|timezone:all';  
  
'timezone' => 'required|timezone:Africa';  
  
'timezone' => 'required|timezone:per_country,US';
```

## unique:table,column

Проверяемое поле не должно существовать в указанной таблице базы данных.

### Указание пользовательского имени таблицы / имени столбца:

Вместо того чтобы указывать имя таблицы напрямую, вы можете указать модель Eloquent, которая должна использоваться для определения имени таблицы:

```
'email' => 'unique:App\Models\User,email_address'
```

Параметр `column` используется для указания соответствующего столбца базы данных поля. Если опция `column` не указана, будет использоваться имя проверяемого поля.

```
'email' => 'unique:users,email_address'
```

### Указание пользовательского соединения базы данных

Иногда требуется указать конкретное соединение для запросов к базе данных, выполняемых валидатором. Для этого вы можете добавить имя подключения к имени таблицы:

```
'email' => 'unique:connection.users,email_address'
```

### Принудительное игнорирование правилом Unique конкретного идентификатора:

Иногда вы можете проигнорировать конкретный идентификатор во время валидации `unique`. Например, рассмотрим страницу «Обновления профиля»,

которая включает имя пользователя, адрес электронной почты и местоположение. Вероятно, вы захотите убедиться, что адрес электронной почты уникален. Однако, если пользователь изменяет только поле имени, а не поле электронной почты, то вы не захотите, чтобы выдавалась ошибка валидации, поскольку пользователь уже является владельцем рассматриваемого адреса электронной почты.

Чтобы указать валидатору игнорировать идентификатор пользователя, мы воспользуемся классом `Rule` для гибкого определения правила. В этом примере мы также укажем правила валидации в виде массива вместо использования символа `|` для их разделения:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

Вы никогда не должны передавать какое-либо введенное пользователем значение из запроса в метод `ignore`. Вместо этого вы должны передавать только сгенерированный системой уникальный идентификатор, такой как автоинкрементный идентификатор или UUID экземпляра модели Eloquent. В противном случае ваше приложение будет уязвимо для атаки с использованием SQL-инъекции.

Вместо того чтобы передавать значение ключа модели методу `ignore`, вы также можете передать весь экземпляр модели. Laravel автоматически извлечет ключ из модели:

```
Rule::unique('users')->ignore($user)
```

Если ваша таблица использует имя столбца с первичным ключом, отличное от `id`, то вы можете указать имя столбца при вызове метода `ignore`:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

По умолчанию правило `unique` проверяет уникальность столбца, совпадающего с именем проверяемого атрибута. Однако вы можете передать другое имя столбца в качестве второго аргумента метода `unique`:

```
Rule::unique('users', 'email_address')->ignore($user->id)
```

### Добавление дополнительных выражений `Where`:

Вы можете указать дополнительные условия запроса, изменив запрос с помощью метода `where`. Например, давайте добавим условие запроса, ограничивающее область запроса только поиском записями, у которых значение столбца `account_id` равно `1`:

```
'email' => Rule::unique('users')->where(fn (Builder $query) => $query->where('account_id', 1))
```

## uppercase

Поле, подлежащее проверке, должно быть в верхнем регистре.

## url

Проверяемое поле должно быть действительным URL.

Если вы хотите указать протоколы URL, которые следует считать допустимыми, вы можете передать их в качестве параметров правила проверки:

```
'url' => 'url:http,https',  
'game' => 'url:minecraft,steam',
```

## ulid

Поле, подлежащее проверке, должно быть допустимым [Универсальным Уникальным Лексикографически Сортируемым Идентификатором](#) (UUID).

## uuid

Проверяемое поле должно быть действительным универсальным уникальным идентификатором (UUID) RFC 4122 (версии 1, 3, 4 или 5).

## # Условное добавление правил

### Пропуск валидации при определенных значениях полей

По желанию можно не проверять конкретное поле, если другое поле имеет указанное значение. Вы можете сделать это, используя правило валидации `exclude_if`. В этом примере поля `appointment_date` и `doctor_name` не будут проверяться, если поле `has_appointment` имеет значение `false`:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_if:has_appointment,false|required|date',
    'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);

```

В качестве альтернативы вы можете использовать правило `exclude_unless`, чтобы не проверять конкретное поле, если другое поле не имеет указанного значения:

```
$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
    'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);

```

### Валидация при условии наличия

По желанию можно выполнить валидацию поля, **только если** это поле присутствует в проверяемых данных. Чтобы этого добиться, добавьте правило `sometimes` в свой список правил:

```
$validator = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);

```

В приведенном выше примере поле `email` будет проверено, только если оно присутствует в массиве `$request->all()`.

Если вы пытаетесь проверить поле, которое всегда должно присутствовать, но может быть пустым, ознакомьтесь с [этим примечанием о необязательных полях](#).

## Комплексная условная проверка

Иногда вы можете добавить правила валидации, основанные на более сложной условной логике. Например, вы можете потребовать обязательного присутствия переданного поля только в том случае, если другое поле имеет значение больше 100. Или вам может потребоваться, чтобы два поля имели указанное значение только при наличии другого поля. Добавление этих правил валидации не должно вызывать затруднений. Сначала создайте экземпляр `Validator` со своими *статическими правилами*, которые будут неизменными:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);

```

Предположим, что наше веб-приложение предназначено для коллекционеров игр. Если коллекционер игр регистрируется в нашем приложении и у него есть более 100 игр, мы хотим, чтобы он объяснил, почему у него так много игр. Например, возможно, они владеют магазином по перепродаже игр или, может быть, им просто нравится коллекционировать игры. Чтобы условно добавить это требование, мы можем использовать метод `sometimes` экземпляра `Validator`:

```
use Illuminate\Support\Fluent;

$validator->sometimes('reason', 'required|max:500', function (Fluent $input) {
    return $input->games >= 100;
});
```

Первый аргумент, переданный методу `sometimes` – это имя поля, которое мы условно проверяем. Второй аргумент – это список правил, которые мы хотим добавить. Если замыкание, переданное в качестве третьего аргумента, возвращает `true`, то правила будут добавлены. Этот метод упрощает создание сложных условных проверок. Вы даже можете добавить условные проверки сразу для нескольких полей:

```
$validator->sometimes(['reason', 'cost'], 'required', function (Fluent $input) {
    return $input->games >= 100;
});
```

Параметр `$input`, переданный вашему замыканию, будет экземпляром `Illuminate\Support\Fluent` и может использоваться при валидации для доступа к вашим входящим данным и файлам запроса.

## Комплексная условная проверка массива

Иногда вам может потребоваться проверить поле на основе другого поля в том же вложенном массиве, индекс которого вам неизвестен. В этих ситуациях вы можете позволить вашему замыканию получить второй аргумент, который будет текущим отдельным элементом в проверяемом массиве:

```
$input = [
    'channels' => [
        [
            'type' => 'email',
            'address' => 'abigail@example.com',
        ],
        [
            'type' => 'url',
            'address' => 'https://example.com',
        ],
    ],
];
```

```
        ],
    ],
];

$validator->sometimes('channels.*.address', 'email', function (Fluent $input, Fluent $item)
{
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function (Fluent $input, Fluent $item)
{
    return $item->type !== 'email';
});
```

Подобно переданному в замыкание параметру `$input`, параметр `$item` является экземпляром `Illuminate\Support\Fluent`, когда значение атрибута является массивом; в противном случае это строка.

## # Валидация массивов

Правило `array`, как это уже обсуждалось [выше](#), принимает список разрешенных ключей массива. Если в массиве присутствуют какие-либо дополнительные ключи, проверка не удастся:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:name,username',
]);
```

В общем, вы всегда должны указывать ключи массива, которые могут присутствовать в вашем массиве. В противном случае методы валидатора `validate` и `validated` вернут все проверенные данные, включая массив и все его ключи, даже если эти ключи не были проверены другими правилами проверки вложенных массивов.

## Проверка входных данных вложенного массива

Проверка полей ввода формы на основе массива не должна быть проблемой. Вы можете использовать «точечную нотацию» для валидации атрибутов в массиве. Например, если входящий HTTP-запрос содержит поле `photos[profile]`, вы можете проверить его следующим образом:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

Вы также можете проверить каждый элемент массива. Например, чтобы убедиться, что каждое электронное письмо в переданном поле ввода массива уникально, вы можете сделать следующее:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Точно так же вы можете использовать символ `*` при указании [пользовательских сообщений в ваших языковых файлах](#), что упрощает использование одного сообщения валидации для полей на основе массива:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique email address',
    ]
],
```

## Доступ к данным вложенного массива

Иногда может возникнуть необходимость получить значение для определенного вложенного элемента массива при назначении правил проверки атрибуту. Это можно сделать с использованием метода `Rule:::forEach`. Метод `forEach` принимает замыкание, которое будет вызываться для каждой итерации массива атрибута подлежащего проверке и получит значение атрибута, а также явное, полностью

расширенное имя атрибута. Замыкание должно возвращать массив правил, которые будут применены к элементу массива:

```
use App\Rules\HasPermission;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$validator = Validator::make($request->all(), [
    'companies.*.id' => Rule::forEach(function (string|null $value, string $attribute) {
        return [
            Rule::exists(Company::class, 'id'),
            new HasPermission('manage-company', $value),
        ];
    }),
]);

```

## Индексы и позиции в сообщениях об ошибках

При валидации массивов может возникнуть необходимость в сообщении об ошибке, отображаемом вашим приложением, ссылаться на индекс или позицию определенного элемента, который не прошел проверку. Для этого можно использовать заполнители :index (начинается с 0) и :position (начинается с 1) в вашем пользовательском сообщении об ошибке:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'photos' => [
        [
            'name' => 'BeachVacation.jpg',
            'description' => 'A photo of my beach vacation!',
        ],
        [
            'name' => 'GrandCanyon.jpg',
            'description' => '',
        ],
    ],
];
Validator::validate($input, [
    'photos.*.description' => 'required',
], [
```

```
'photos.*.description.required' => 'Пожалуйста, укажите описание для фото № :pos:  
]);
```

На основе приведенного выше примера валидация завершится неудачей, и пользователю будет представлена следующая ошибка: "Пожалуйста, укажите описание для фото №2."

При необходимости вы можете ссылаться на более глубокие вложенные индексы и позиции с использованием `second-index`, `second-position`, `third-index`, `third-position` и так далее.

```
'photos.*.attributes.*.string' => 'Invalid attribute for photo #:second-position.',
```



## # Валидация файлов

Laravel предоставляет различные правила валидации, которые можно использовать для проверки загруженных файлов, такие как `mimes`, `image`, `min` и `max`. Хотя вы можете указать эти правила индивидуально при валидации файлов, Laravel также предлагает удобный конструктор правил валидации файлов:

```
use Illuminate\Support\Facades\Validator;  
use Illuminate\Validation\Rules\File;  
  
Validator::validate($input, [  
    'attachment' => [  
        'required',  
        File::types(['mp3', 'wav'])  
            ->min(1024)  
            ->max(12 * 1024),  
    ],  
]);
```

Если ваше приложение принимает изображения, загруженные пользователями, вы можете использовать метод `image` правила `File`, чтобы указать, что загруженный файл должен быть изображением. Кроме того, правило `dimensions` может быть использовано для ограничения размеров изображения:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'photo' => [
        'required',
        File::image()
            ->min(1024)
            ->max(12 * 1024)
            ->dimensions(Rule::dimensions()->maxWidth(1000)->maxHeight(500)),
    ],
]);
```

Дополнительную информацию о валидации размеров изображения можно найти в [документации по правилу dimensions](#).

## Размеры файлов

Для удобства минимальные и максимальные размеры файлов могут быть указаны в виде строки с суффиксом, указывающим единицы измерения размера файла.

Поддерживаются суффиксы `kb`, `mb`, `gb` и `tb`:

```
File::image()
->min('1kb')
->max('10mb')
```

## Типы файлов

Несмотря на то, что вам нужно указать только расширения при вызове метода `types`, этот метод фактически проверяет MIME-тип файла, считывая содержимое файла и угадывая его MIME-тип. Полный список MIME-типов и их соответствующих расширений можно найти по следующему адресу:

<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

## # Валидация паролей

Чтобы гарантировать, что пароли имеют достаточный уровень сложности, вы можете использовать объект правила laravel `Password`:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [
    'password' => ['required', 'confirmed', Password::min(8)],
]);
```

Объект правила `Password` позволяет вам легко настроить требования к сложности пароля для вашего приложения, например указать, что для паролей требуется хотя бы одна буква, цифра, символ или символы со смешанным регистром:

```
// Требуется не менее 8 символов ...
Password::min(8)

// Требуется хотя бы одна буква ...
Password::min(8)->letters()

// Требуется хотя бы одна заглавная и одна строчная буква...
Password::min(8)->mixedCase()

// Требовать хотя бы одна цифра...
Password::min(8)->numbers()

// Требуется хотя бы один символ...
Password::min(8)->symbols()
```

Кроме того, вы можете убедиться, что пароль не был скомпрометирован в результате утечки данных публичного пароля, используя метод `uncompromised`:

```
Password::min(8)->uncompromised()
```

Объект правила `Password` использует модель [k-Anonymity](#), чтобы определить, не произошла ли утечка пароля через [haveibeenpwned.com](http://haveibeenpwned.com) без ущерба для конфиденциальности или безопасности пользователя.

По умолчанию, если пароль появляется хотя бы один раз при утечке данных, он считается скомпрометированным. Вы можете настроить этот порог, используя первый аргумент метода `uncompromised`:

```
// Убедитесь, что пароль появляется не реже 3 раз в одной и той же утечке данных...
Password::min(8)->uncompromised(3);
```

Конечно, вы можете связать все методы в приведенных выше примерах:

```
Password::min(8)
->letters()
->mixedCase()
->numbers()
->symbols()
->uncompromised()
```

## Определение правил паролей по умолчанию

Возможно, вам будет удобно указать правила проверки паролей по умолчанию в одном месте вашего приложения. Вы можете легко сделать это, используя метод `Password::defaults`, который принимает замыкание. Замыкание, данное методу `defaults`, должно вернуть конфигурацию правила пароля по умолчанию. Чаще всего, правило `defaults` следует вызывать в методе `boot` одного из поставщиков услуг вашего приложения:

```
use Illuminate\Validation\Rules\Password;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Password::defaults(function () {
        $rule = Password::min(8);

        return $this->app->isProduction()
            ? $rule->mixedCase()->uncompromised()
            : $rule;
    });
}
```

Затем, когда вы хотите применить правила по умолчанию к конкретному паролю, проходящему проверку, вы можете вызвать метод `defaults` без аргументов:

```
'password' => ['required', Password::defaults()],
```

Иногда вы можете захотеть добавить дополнительные правила проверки к правилам проверки пароля по умолчанию. Для этого вы можете использовать метод `rules`: use App\Rules\ZxcvbnRule;

```
Password::defaults(function () {
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

## # Пользовательские правила валидации

### Использование класса Rule

Laravel предлагает множество полезных правил валидации; однако вы можете указать свои собственные. Один из методов регистрации собственных правил валидации – использование объектов правил. Чтобы сгенерировать новый объект правила, вы можете использовать команду `make:rule Artisan`. Давайте воспользуемся этой командой, чтобы сгенерировать правило, которое проверяет, что строка состоит из прописных букв. Laravel поместит новый класс правила в каталог `app/Rules` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его, когда вы выполните команду Artisan для создания своего правила:

```
php artisan make:rule Uppercase
```

Как только правило создано, мы готовы определить его поведение. Объект правила содержит единственный метод: `validate`. Этот метод принимает имя атрибута, его значение и обратный вызов, который должен быть вызван в случае ошибки с сообщением об ошибке валидации:

```
<?php

namespace App\Rules;

use Closure;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements ValidationRule
{
    /**
     * Запустить правило проверки.
     */
    public function validate(string $attribute, mixed $value, Closure $fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail('The :attribute must be uppercase.');
        }
    }
}
```

После определения правила вы можете отправить его валидатору, передав экземпляр объекта правила с другими вашими правилами валидации:

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

## Перевод сообщений валидации

Вместо предоставления буквального сообщения об ошибке в замыкание `$fail`, вы также можете указать [ключ строки перевода](#) и указать Laravel перевести сообщение об ошибке:

```
if (strtoupper($value) !== $value) {
    $fail('validation.uppercase')->translate();
}
```

При необходимости вы можете предоставить замены заполнителей и предпочтительный язык в качестве первого и второго аргументов для

метода `translate`:

```
$fail('validation.location')->translate([
    'value' => $this->value,
], 'fr')
```

## Доступ к дополнительным данным

Если вашему пользовательскому классу правил проверки требуется доступ ко всем другим данным, проходящим проверку, ваш класс правил может реализовать интерфейс `Illuminate\Contracts\Validation\DataAwareRule`. Этот интерфейс требует, чтобы ваш класс определил метод `setData`. Этот метод будет автоматически вызван Laravel (до того, как начнется проверка) со всеми проверяемыми данными:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements DataAwareRule, ValidationRule
{
    /**
     * All of the data under validation.
     *
     * @var array<string, mixed>
     */
    protected $data = [];

    // ...

    /**
     * Set the data under validation.
     *
     * @param array<string, mixed> $data
     */
    public function setData(array $data): static
    {
        $this->data = $data;

        return $this;
    }
}
```

Или, если ваше правило проверки требует доступа к экземпляру валидатора, выполняющему проверку, вы можете реализовать интерфейс [ValidatorAwareRule](#):

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
use Illuminate\Validation\Validator;

class Uppercase implements ValidationRule, ValidatorAwareRule
{
    /**
     * The validator instance.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * Set the current validator.
     */
    public function setValidator(Validator $validator): static
    {
        $this->validator = $validator;

        return $this;
    }
}
```

## Использование замыканий

Если вам нужна функциональность собственного правила только один раз во всем приложении, то вы можете использовать анонимное правило вместо объекта правила. Анонимное правило получит имя атрибута, значение атрибута и замыкание [\\$fail](#), которое будет выполнено в случае провала проверки:

```
use Illuminate\Support\Facades\Validator;
use Closure;

$validator = Validator::make($request->all(), [
    'title' => [
```

```
'required',
'max:255',
function (string $attribute, mixed $value, Closure $fail) {
    if ($value === 'foo') {
        $fail("The {$attribute} is invalid.");
    }
},
],
]);
});
```

## Неявные правила

По умолчанию, правила валидации, включая созданные вами, не применяются, если проверяемый атрибут отсутствует или содержит пустую строку. Например, правило `unique` не будет выполнено для пустой строки:

```
use Illuminate\Support\Facades\Validator;

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true
```

Чтобы ваше правило было применено, даже если атрибут пуст, то правило должно подразумевать, что атрибут является обязательным. Чтобы быстрого сгенерировать новый объект неявного правила, вы можете использовать Artisan-команду `make:rule` с параметром `--implicit`:

```
php artisan make:rule Uppercase --implicit
```

«Неявное» правило только *подразумевает*, что атрибут является обязательным к валидации. В действительности решать только вам, будет ли пустой или отсутствующий атрибут считаться невалидным.

# Обработка ошибок (Exception)

# Введение

# Конфигурирование

# Обработка исключений

# Отчет об исключениях

# Уровни журнала исключений

# Игнорирование исключений по типу

# Отображение исключений

# Отчетные и отображаемые исключения

# Ограничение на количество зарегистрированных исключений

# HTTP-исключения

# Пользовательские страницы для HTTP ошибок

## # Введение

Когда вы запускаете новый проект Laravel, обработка ошибок и исключений уже настроена для вас; однако в любой момент вы можете использовать метод `withExceptions` в файле `bootstrap/app.php` вашего приложения, чтобы управлять тем, как ваше приложение сообщает об исключениях и обрабатывает их.

Объект `$exceptions`, предоставляемый замыканием `withExceptions`, является экземпляром `Illuminate\Foundation\Configuration\Exceptions` и отвечает за управление обработкой исключений в вашем приложении. В этой документации мы углубимся в этот объект.

## # Конфигурирование

Параметр `debug` в конфигурационном файле `config/app.php` определяет, сколько информации об ошибке фактически отобразится пользователю. По умолчанию этот параметр установлен, чтобы учесть значение переменной окружения `APP_DEBUG`, которая содержится в вашем файле `.env`.

Во время локальной разработки вы должны установить для переменной окружения `APP_DEBUG` значение `true`. Во время эксплуатации приложения это значение всегда должно быть `false`. Если в рабочем окружении будет установлено значение `true`, вы рискуете раскрыть конфиденциальные значения конфигурации конечным пользователям вашего приложения.

## # Обработка исключений

### Отчет об исключениях

В Laravel отчеты об исключениях используются для регистрации исключений или их отправки во внешнюю службу [Sentry](#) или [Flare](#). По умолчанию исключения будут регистрироваться на основе вашей конфигурации [logging](#). Однако вы можете зарегистрировать исключения по своему усмотрению.

Если вам нужно сообщать о различных типах исключений разными способами, вы можете использовать метод исключений `report` в файле `bootstrap/app.php` вашего приложения, чтобы зарегистрировать замыкание, которое должно выполняться, когда необходимо сообщить об исключении определенного типа. Laravel определит, о каком типе исключения сообщает замыкание, исследуя подсказку типа замыкания:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->report(function (InvalidOrderException $e) {
        // ...
    });
})
```

Когда вы регистрируете собственные замыкания для создания отчетов об исключениях, используя метод `report`, Laravel по-прежнему регистрирует исключение, используя конфигурацию логирования по умолчанию для приложения. Если вы хотите остановить распространение исключения в стек журналов по умолчанию, вы можете использовать метод `stop` при определении замыкания отчета или вернуть `false` из замыкания:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->report(function (InvalidOrderException $e) {
        // ...
    })->stop();
})
```

```
$exceptions->report(function (InvalidOrderException $e) {
    return false;
});
})
```

Чтобы настроить отчет об исключениях для переданного исключения, вы можете рассмотреть возможность использования [отчетных исключений](#).

## Глобальное содержимое журнала

Если доступно, Laravel автоматически добавляет идентификатор текущего пользователя в каждое сообщение журнала исключения в качестве контекстных данных. Вы можете определить свои собственные глобальные контекстные данные, используя метод исключения `context` в файле `bootstrap/app.php` вашего приложения. Эта информация будет включена в каждое сообщение журнала исключения, записанное вашим приложением:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->context(fn () => [
        'foo' => 'bar',
    ]);
})
```

## Контекст журнала исключений

Добавление контекста к каждому сообщению в журнале может быть полезным, но иногда у конкретного исключения может быть уникальный контекст, который вы хотели бы включить в журнал. Определив метод `context` в одном из исключений вашего приложения, вы можете указать любые данные, относящиеся к этому исключению, которые должны быть добавлены в журнал записи об исключении:

```
<?php

namespace App\Exceptions;

use Exception;
```

```
class InvalidOrderException extends Exception
{
    // ...

    /**
     * Получить контекстную информацию исключения.
     *
     * @return array<string, mixed>
     */
    public function context(): array
    {
        return ['order_id' => $this->orderId];
    }
}
```

## Помощник report

По желанию может потребоваться сообщить об исключении, но продолжить обработку текущего запроса. Помощник `report` позволяет вам быстро сообщить об исключении, не отображая страницу с ошибкой для пользователя:

```
public function isValid(string $value): bool
{
    try {
        // Проверка `$value` ...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
}
```

## Исключения дубликатов

Если вы используете функцию `report` в вашем приложении, вы иногда можете сообщать об одном и том же исключении несколько раз, создавая дублирующие записи в журналах.

Если вы хотите, чтобы об одном экземпляре исключения сообщалось только один раз, вы можете вызвать метод исключения `dontReportDuplicates` в файле `bootstrap/app.php` вашего приложения:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->dontReportDuplicates();
})
```

Теперь, когда функция `report` вызывается с тем же экземпляром исключения, будет сообщено только первое вызов:

```
$original = new RuntimeException('Whoops!');

report($original); // сообщено

try {
    throw $original;
} catch (Throwable $caught) {
    report($caught); // проигнорировано
}

report($original); // проигнорировано
report($caught); // проигнорировано
```

## Уровни журнала исключений

Когда сообщения записываются в [журнал вашего приложения](#), сообщения записываются с указанным [уровнем журнала](#), который указывает на серьезность или важность сообщения, которое записывается.

Как отмечено выше, даже когда вы регистрируете пользовательский обратный вызов сообщения об исключении с использованием метода `report`, Laravel все равно будет записывать исключение с использованием конфигурации регистрации журнала по умолчанию для приложения. Однако поскольку уровень журнала иногда может влиять на каналы, на которых записывается сообщение, вы можете настроить уровень журнала, на котором определенные исключения записываются.

Для этого вы можете использовать метод исключения `level` в файле `bootstrap/app.php` вашего приложения. Этот метод получает тип исключения в качестве первого аргумента и уровень журнала в качестве второго аргумента:

```
use PDOException;
use Psr\Log\LogLevel;

->withExceptions(function (Exceptions $exceptions) {
```

```
$exceptions->level(PDOException::class, LogLevel::CRITICAL);  
})
```

## Игнорирование исключений по типу

При создании приложения могут возникнуть некоторые типы исключений, о которых вы никогда не захотите сообщать. Чтобы игнорировать эти исключения, вы можете использовать метод исключений `dontReport` в файле `bootstrap/app.php` вашего приложения. Ни о каком классе, предоставленном этому методу, никогда не будет сообщено; однако они все равно могут иметь собственную логику рендеринга:

```
use App\Exceptions\InvalidOrderException;  
  
->withExceptions(function (Exceptions $exceptions) {  
    $exceptions->dontReport([  
        InvalidOrderException::class,  
    ]);  
})
```

В качестве альтернативы вы можете просто «пометить» класс исключений с помощью интерфейса `Illuminate\Contracts\Debug\ShouldntReport`. Когда исключение помечено этим интерфейсом, обработчик исключений Laravel никогда не сообщит об этом:

```
<?php  
  
namespace App\Exceptions;  
  
use Exception;  
use Illuminate\Contracts\Debug\ShouldntReport;  
  
class PodcastProcessingException extends Exception implements ShouldntReport  
{  
    //  
}
```

Внутри Laravel уже игнорирует некоторые типы ошибок, например исключения, возникающие из-за ошибок 404 HTTP или ответов 419 HTTP, сгенерированных недействительными токенами CSRF. Если вы хотите указать Laravel прекратить

игнорировать определенный тип исключения, вы можете использовать метод исключения `stopIgnoring` в файле `bootstrap/app.php` вашего приложения:

```
use Symfony\Component\HttpKernel\Exception\HttpException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->stopIgnoring(HttpException::class);
})
```

## Отображение исключений

По умолчанию обработчик исключений Laravel преобразует исключения в HTTP-ответ. Однако вы можете зарегистрировать свое замыкание для исключений заданного типа. Вы можете добиться этого, используя метод исключения `render` в файле `bootstrap/app.php` вашего приложения.

Замыкание, переданное методу `render`, должно вернуть экземпляр `Illuminate\Http\Response`, который может быть сгенерирован с помощью функции `response`. Laravel определит, какой тип исключения отображает замыкание с помощью типизации аргументов:

```
use App\Exceptions\InvalidOrderException;
use Illuminate\Http\Request;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->render(function (InvalidOrderException $e, Request $request) {
        return response()->view('errors.invalid-order', status: 500);
    });
})
```

Вы также можете использовать метод `render` чтобы переопределить отображение для встроенных исключений Laravel или Symfony, таких, как `NotFoundHttpException`. Если замыкание, переданное методу `render` не возвращает значения, будет использоваться отрисовка исключений Laravel по умолчанию:

```
use Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->render(function (NotFoundHttpException $e, Request $request) {
        if ($request->is('api/*')) {
```

```
        return response()->json([
            'message' => 'Record not found.'
        ], 404);
    }
});

})
```

## Отображение исключений в формате JSON

When rendering an exception, Laravel will automatically determine if the exception should be rendered as an HTML or JSON response based on the `Accept` header of the request. If you would like to customize how Laravel determines whether to render HTML or JSON exception responses, you may utilize the `shouldRenderJsonWhen` method: При обработке исключения Laravel автоматически определяет, должно ли исключение быть отображено в виде ответа HTML или JSON, на основе заголовка `Accept` запроса. Если вы хотите настроить, как Laravel определяет, следует ли отображать ответы об исключениях HTML или JSON, вы можете использовать метод `shouldRenderJsonWhen`:

```
use Illuminate\Http\Request;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->shouldRenderJsonWhen(function (Request $request, Throwable $e) {
        if ($request->is('admin/*')) {
            return true;
        }

        return $request->expectsJson();
    });
})
```

## Настройка ответа на исключение

В редких случаях вам может потребоваться настроить весь HTTP-ответ, отображаемый обработчиком исключений Laravel. Для этого вы можете зарегистрировать закрытие настройки ответа, используя метод `respond`:

```
use Symfony\Component\HttpFoundation\Response;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->respond(function (Response $response) {
```

```
        if ($response->getStatusCode() === 419) {
            return back()->with([
                'message' => 'The page expired, please try again.',
            ]);
        }

        return $response;
    });
}

)
```

## Отчетные и отображаемые исключения

Вместо того чтобы настраивать пользовательское поведение отчетов и отображение ошибок в файле `bootstrap/app.php` вашего приложения, вы можете определить методы `report` и `render` непосредственно в самих классах исключений вашего приложения. Когда эти методы существуют, фреймворк автоматически будет вызывать их для обработки ошибок:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Http\Request;
use Illuminate\Http\Response;

class InvalidOrderException extends Exception
{
    /**
     * Отчитаться об исключении.
     */
    public function report() : void
    {
        // ...
    }

    /**
     * Преобразовать исключение в HTTP-ответ.
     */
    public function render(Request $request): Response
    {
        return response(/* ... */);
    }
}
```

Если ваше исключение расширяет исключение, которое уже доступно для визуализации, например встроенное исключение Laravel или Symfony, вы можете вернуть `false` из метода `render` исключения, чтобы отобразить HTTP-ответ исключения по умолчанию:

```
/*
 * Преобразовать исключение в HTTP-ответ.
 */
public function render(Request $request): Response|bool
{
    if (/** Определить, требуется ли для исключения пользовательское отображение */)

        return response(/* ... */);
    }

    return false;
}
```

Если ваше исключение содержит пользовательскую логику отчетности, которая необходима только при выполнении определенных условий, то вам может потребоваться указать Laravel когда сообщать об исключении, используя конфигурацию обработки исключений по умолчанию. Для этого вы можете вернуть `false` из метода `report` исключения:

```
/*
 * Сообщить об исключении.
 */
public function report(): bool
{
    if (/** Определить, требуется ли для исключения пользовательское отображение */)

        return true;
    }

    return false;
}
```

Вы можете указать любые требуемые зависимости метода `report`, и они будут автоматически внедрены

в метод [контейнером служб](#) Laravel.

## Ограничение на количество зарегистрированных исключений

Если ваше приложение регистрирует очень большое количество исключений, вам может потребоваться ограничить количество фактически регистрируемых или отправляемых во внешний сервис отслеживания ошибок.

Чтобы получить случайную частоту выборки исключений, вы можете использовать метод исключений `throttle` в файле `bootstrap/app.php` вашего приложения. Метод `throttle` получает замыкание, которое должно возвращать экземпляр `Lottery`:

```
use Illuminate\Support\Lottery;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        return Lottery::odds(1, 1000);
    });
})
```

Также можно условно выбирать исключения на основе их типа. Если вы хотите выбирать только экземпляры конкретного класса исключений, вы можете вернуть экземпляр `Lottery` только для этого класса:

```
use App\Exceptions\ApiMonitoringException;
use Illuminate\Support\Lottery;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof ApiMonitoringException) {
            return Lottery::odds(1, 1000);
        }
    });
})
```

Вы также можете ограничивать количество исключений, зарегистрированных или отправленных во внешний сервис отслеживания ошибок, вернув экземпляр `Limit`

вместо [Lottery](#). Это полезно, если вы хотите защититься от внезапных всплесков исключений, засоряющих ваши логи, например, когда сторонний сервис, используемый вашим приложением, недоступен:

```
use Illuminate\\Broadcasting\\BroadcastException;
use Illuminate\\Cache\\RateLimiting\\Limit;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof BroadcastException) {
            return Limit::perMinute(300);
        }
    });
})
```

По умолчанию ограничения будут использовать класс исключения в качестве ключа ограничения по количеству. Вы можете настроить это, указав свой собственный ключ с помощью метода [by](#) на [Limit](#):

```
use Illuminate\\Broadcasting\\BroadcastException;
use Illuminate\\Cache\\RateLimiting\\Limit;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
        if ($e instanceof BroadcastException) {
            return Limit::perMinute(300)->by($e->getMessage());
        }
    });
})
```

Конечно же, вы можете возвращать смешанные экземпляры [Lottery](#) и [Limit](#) для разных исключений:

```
use App\\Exceptions\\ApiMonitoringException;
use Illuminate\\Broadcasting\\BroadcastException;
use Illuminate\\Cache\\RateLimiting\\Limit;
use Illuminate\\Support\\Lottery;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->throttle(function (Throwable $e) {
```

```
        return match (true) {
            $e instanceof BroadcastException => Limit::perMinute(300),
            $e instanceof ApiMonitoringException => Lottery::odds(1, 1000),
            default => Limit::none(),
        };
    });
}
});
```

## # HTTP-исключения

Некоторые исключения описывают коды HTTP-ошибок с сервера. Например, это может быть ошибка «страница не найдена» (404), «неавторизованный доступ» (401) или даже ошибка 500, сгенерированная разработчиком. Чтобы создать такой ответ из любой точки вашего приложения, вы можете использовать глобальный помощник `abort`:

```
abort(404);
```

## Пользовательские страницы для HTTP ошибок

Laravel позволяет легко отображать пользовательские страницы ошибок для различных кодов состояния HTTP. Например, если вы хотите настроить страницу ошибок для кодов HTTP-состояния 404, создайте файл `resources/views/errors/404.blade.php`. Это представление будет отображено для всех ошибок 404, сгенерированных вашим приложением. Шаблоны в этом каталоге должны быть названы в соответствии с кодом состояния HTTP, которому они соответствуют. Экземпляр `Symfony\Component\HttpKernel\Exception\HttpException`, вызванный функцией `abort`, будет передан в шаблон как переменная `$exception`:

```
<h2>{{ $exception->getMessage() }}</h2>
```

Вы можете опубликовать стандартные шаблоны страниц ошибок Laravel с помощью команды `vendor:publish` Artisan. После публикации шаблонов вы можете настроить их по своему вкусу:

```
php artisan vendor:publish --tag=laravel-errors
```

## Запасные страницы для HTTP ошибок

Вы также можете определить “запасную” страницу ошибки для определенного набора кодов состояния HTTP. Эта страница будет отображаться, если нет соответствующей страницы для конкретного кода состояния HTTP, который произошел. Для этого определите шаблон `4xx.blade.php` и шаблон `5xx.blade.php` в директории `resources/views/errors` вашего приложения.

При определении “запасных” страниц ошибок, “запасные” страницы не влияют на ответы об ошибках `404`, `500` и `503`, поскольку в Laravel есть внутренние, выделенные страницы для этих кодов состояния. Чтобы настроить страницы, отображаемые для этих кодов состояния, необходимо определить собственную страницу ошибок для каждого из них индивидуально.

# Логирование

## # Введение

## # Конфигурирование

- # Доступные драйверы канала
- # Предварительная подготовка канала
- # Логирование предупреждений об устаревании

## # Построение стека журналов

## # Запись сообщений журнала

- # Контекстная информация
- # Запись в определенные каналы

## # Настройка канала Monolog

- # Настройка Monolog для каналов
- # Создание обработчика каналов Monolog
- # Создание каналов через фабрики

## # Просмотр сообщения журнала с помощью Pail

- # Установка
- # Использование
- # Фильтрация логов

## # Введение

Чтобы помочь вам узнать больше о том, что происходит в вашем приложении, Laravel предлагает надежные службы ведения журнала, которые позволяют записывать сообщения в файлы, журнал системных ошибок и даже в Slack, чтобы уведомить всю вашу команду.

Ведение журнала Laravel основано на «каналах». Каждый канал представляет собой определенный способ записи информации журнала. Например, канал `single` записывает файлы журнала в один файл журнала, а канал `slack` отправляет сообщения журнала в Slack. Сообщения журнала могут быть записаны в несколько каналов в зависимости от их серьезности.

Под капотом Laravel использует библиотеку [Monolog](#), которая обеспечивает поддержку множества мощных обработчиков журналов. Laravel упрощает настройку этих обработчиков, позволяя вам смешивать и сопоставлять их для настройки обработки журналов вашего приложения.

## # Конфигурирование

Все параметры конфигурации, которые управляют ведением журнала вашего приложения размещены в файле конфигурации `config/logging.php`. Этот файл позволяет вам настраивать каналы журнала вашего приложения, поэтому обязательно просмотрите каждый из доступных каналов и их параметры. Ниже мы рассмотрим несколько распространенных вариантов.

По умолчанию Laravel будет использовать канал `stack` при регистрации сообщений. Канал `stack` используется для объединения нескольких каналов журнала в один канал. Для получения дополнительной информации о построении стеков ознакомьтесь с [документацией ниже](#).

## Доступные драйверы канала

Каждый канал журнала работает через «драйвер». Драйвер определяет, как и где фактически записывается сообщение журнала. Следующие драйверы канала журнала доступны в каждом приложении Laravel. Запись для большинства этих драйверов уже присутствует в файле конфигурации вашего приложения `config/logging.php`, поэтому обязательно просмотрите этот файл, чтобы ознакомиться с его содержимым:

Имя	Описание
<code>custom</code>	Драйвер, который вызывает указанную фабрику для создания канала.
<code>daily</code>	Драйвер Monolog на основе <code>RotatingFileHandler</code> с ежедневной ротацией.
<code>errorlog</code>	Драйвер Monolog на основе <code>ErrorLogHandler</code> .

Имя	Описание
<code>monolog</code>	Драйвер фабрики Monolog, использующий любой поддерживаемый Monolog обработчик.
<code>papertrail</code>	Драйвер Monolog на основе <code>SyslogUdpHandler</code> .
<code>single</code>	Канал на основе одного файла или пути ( <code>StreamHandler</code> )
<code>slack</code>	Драйвер Monolog на основе <code>SlackWebhookHandler</code> .
<code>stack</code>	Обертка для облегчения создания «многоканальных» каналов.
<code>syslog</code>	Драйвер Monolog на основе <code>SyslogHandler</code> .

Ознакомьтесь с документацией по [продвинутой  
кастомизации каналов](#), чтобы узнать больше о драйверах `monolog` и `custom`.

## Настройка имени канала

По умолчанию экземпляр Monolog создается с «именем канала», которое соответствует текущей среде, например, `production` или `local`. Чтобы изменить это значение, добавьте параметр `name` в конфигурацию вашего канала:

```
'stack' => [  
    'driver' => 'stack',  
    'name' => 'channel-name',  
    'channels' => ['single', 'slack'],  
,
```

## Предварительная подготовка канала

### Конфигурирование каналов Single и Daily

Каналы (Channels) `single` и `daily` имеют три необязательных параметра конфигурации: `bubble`, `permission`, и `locking`.

Имя	Описание	По умолчанию
<code>bubble</code>	Должны ли сообщения переходить в другие каналы после обработки	<code>true</code>
<code>locking</code>	Попытаться заблокировать файл журнала перед записью в него	<code>false</code>
<code>permission</code>	Права доступа на файл журнала	<code>0644</code>

Дополнительно, способ хранения для канала `daily` можно настроить с помощью переменной среды `LOG_DAILY_DAYS` или путем установки параметра конфигурации `days`.

Name	Description	Default
<code>days</code>	Количество дней, в течение которых следует хранить файлы daily channel.	<code>7</code>

## Конфигурирование канала Papertrail

Для канала `papertrail` требуются параметры конфигурации `host` и `port`. Их можно определить с помощью переменных среды `PAPERTRAIL_URL` и `PAPERTRAIL_PORT`. Эти значения можно получить из [Papertrail](#).

## Конфигурирование канала Slack

Для канала `slack` требуется параметр конфигурации `url`. Это значение может быть определено через переменную среды `LOG_SLACK_WEBHOOK_URL`. Этот URL-адрес должен соответствовать URL-адресу [входящего веб-хука](#), который вы настроили для своей команды Slack.

По умолчанию Slack будет получать логи только с уровнем `critical` и выше; однако вы можете настроить это, используя переменную среды `LOG_LEVEL` или изменив параметр конфигурации `level` в массиве вашего драйвера Slack.

# Логирование предупреждений об устаревании

PHP, Laravel и другие библиотеки часто уведомляют своих пользователей о том, что некоторые из их функций устарели и будут удалены в будущей версии. Если вы хотите регистрировать эти предупреждения об устаревании, вы можете указать предпочитаемый канал журнала `deprecations`, используя переменную среды `LOG_DEPRECATED_CHANNEL` или в файле конфигурации вашего приложения `config/logging.php`:

```
'deprecations' => [
    'channel' => env('LOG_DEPRECATED_CHANNEL', 'null'),
    'trace' => env('LOG_DEPRECATED_TRACE', false),
],  
  
'channels' => [
    // ...
]
```

Или вы можете определить канал журнала с именем `deprecations`. Если канал журнала с таким именем существует, он всегда будет использоваться для регистрации устаревания:

```
'channels' => [
    'deprecations' => [
        'driver' => 'single',
        'path' => storage_path('logs/php-deprecation-warnings.log'),
    ],
],
```

## # Построение стека журналов

Как упоминалось ранее, драйвер `stack` позволяет для удобства объединить несколько каналов в один канал журнала. Чтобы проиллюстрировать, как использовать стеки журналов, давайте рассмотрим пример конфигурации, которую вы можете увидеть в эксплуатационном приложении:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'], // [tl! add]
```

```
'ignore_exceptions' => false,  
],  
  
'syslog' => [  
    'driver' => 'syslog',  
    'level' => env('LOG_LEVEL', 'debug'),  
    'facility' => env('LOG_SYSLOGFacility', LOG_USER),  
    'replace_placeholders' => true,  
],  
  
'slack' => [  
    'driver' => 'slack',  
    'url' => env('LOG_SLACK_WEBHOOK_URL'),  
    'username' => env('LOG_SLACK_USERNAME', 'Laravel Log'),  
    'emoji' => env('LOG_SLACK_EMOJI', ':boom:'),  
    'level' => env('LOG_LEVEL', 'critical'),  
    'replace_placeholders' => true,  
],  
],
```

Давайте разберем эту конфигурацию. Во-первых, обратите внимание, что наш канал `stack` объединяет два других канала с помощью параметра `channels: syslog` и `slack`. Таким образом, при регистрации сообщений оба канала будут иметь возможность регистрировать сообщение. Однако, как мы увидим ниже, действительно ли эти каналы регистрируют сообщение, может быть определено серьезностью / «уровнем» сообщения.

## Уровни журнала

Обратите внимание на параметр конфигурации `level`, присутствующий в конфигурациях каналов `syslog` и `slack` в приведенном выше примере. Эта опция определяет минимальный «уровень» сообщения, которое должно быть зарегистрировано каналом. Monolog, на котором работают службы ведения журналов Laravel, предлагает все уровни журналов, определенные в спецификации [RFC 5424 specification](#). Эти уровни журнала в порядке убывания критичности: `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info`, и `debug`.

Итак, представьте, что мы регистрируем сообщение, используя метод `debug`:

```
Log::debug('An informational message.');
```

Учитывая нашу конфигурацию, канал `syslog` будет записывать сообщение в системный журнал; однако, поскольку сообщение об ошибке не является уровнем `critical` или выше, то оно не будет отправлено в Slack. Однако, если мы регистрируем сообщение уровня `emergency`, то оно будет отправлено как в системный журнал, так и в Slack, поскольку уровень `emergency` выше нашего минимального порогового значения для обоих каналов:

```
Log::emergency('The system is down!');
```

## # Запись сообщений журнала

Вы можете записывать информацию в журналы с помощью [Фасада Log](#). Как упоминалось ранее, средство ведения журнала обеспечивает восемь уровней ведения журнала, определенных в спецификации [RFC 5424 specification](#): `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info`, и `debug`.

```
use Illuminate\Support\Facades\Log;

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

Вы можете вызвать любой из этих методов, чтобы записать сообщение для соответствующего уровня. По умолчанию сообщение будет записано в канал журнала по умолчанию, как настроено вашим файлом конфигурации `logging`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Log;
use Illuminate\View\View;
```

```
class UserController extends Controller
{
    /**
     * Показать профиль конкретного пользователя.
     *
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function show(string $id): View
    {
        Log::info('Showing the user profile for user: {id}', ['id' => $id]);

        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

## Контекстная информация

Методам журнала может быть передан массив контекстных данных. Эти контекстные данные будут отформатированы и отображены в сообщении журнала:

```
use Illuminate\Support\Facades\Log;

Log::info('User {id} failed to login.', ['id' => $user->id]);
```

Иногда вы можете указать некоторую контекстную информацию, которая должна быть включена во все последующие записи журнала в определенном канале. Например, вы можете захотеть зарегистрировать идентификатор запроса, связанный с каждым входящим запросом к вашему приложению. Для этого вы можете вызвать метод `withContext` фасада `Log`:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
```

```

{
    /**
     * Обработчик входящего запроса .
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::withContext([
            'request-id' => $requestId
        ]);

        $response = $next($request);

        $response->headers->set('Request-Id', $requestId);

        return $response;
    }
}

```

Если вы хотите добавить общую информацию между *всеми* каналами, вы можете вызвать метод `Log::shareContext()`. Этот метод предоставит дополнительную информацию всем созданным каналам и всем каналам, которые будут созданы впоследствии.

```

<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;

class AssignRequestId
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */

```

```
public function handle(Request $request, Closure $next): Response
{
    $requestId = (string) Str::uuid();

    Log::shareContext([
        'request-id' => $requestId
    ]);

    // ...
}

}
```

Если вам нужно передавать контекст журнала при обработке задач в очереди, вы можете использовать [middleware заданий](#).

## Запись в определенные каналы

По желанию можно записать сообщение в канал, отличный от канала по умолчанию вашего приложения. Вы можете использовать метод `channel` фасада `Log` для получения и регистрации любого канала, определенного в вашем файле конфигурации:

```
use Illuminate\Support\Facades\Log;

Log::channel('slack')->info('Something happened!');
```

Если вы хотите создать стек протоколирования по запросу, состоящий из нескольких каналов, вы можете использовать метод `stack`:

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

## Каналы по запросу

Также возможно создать канал по запросу, предоставив конфигурацию во время выполнения, без того, чтобы эта конфигурация присутствовала в файле `logging`

вашего приложения. Для этого вы можете передать массив конфигурации методу `build` фасада `Log`:

```
use Illuminate\Support\Facades\Log;

Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
])->info('Something happened!');
```

Вы также можете включить канал по запросу в стек журналов по запросу. Этого можно добиться, включив экземпляр вашего канала по запросу в массив, переданный в метод `stack`:

```
use Illuminate\Support\Facades\Log;

$channel = Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
]);

Log::stack(['slack', $channel])->info('Something happened!');
```

## # Настройка канала Monolog

### Настройка Monolog для каналов

Иногда требуется полный контроль над настройкой Monolog для существующего канала. Например, бывает необходимо настроить собственную реализацию Monolog `FormatterInterface` для встроенного в Laravel канала `single`.

Для начала определите массив `tap` в конфигурации канала. Массив `tap` должен содержать список классов, которые должны иметь возможность настраивать (или «касаться») экземпляр Monolog после его создания. Не существует обычного места для размещения этих классов, поэтому вы можете создать каталог в своем приложении, чтобы разместить эти классы:

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
```

```
'path' => storage_path('logs/laravel.log'),
'level' => env('LOG_LEVEL', 'debug'),
'replace_placeholders' => true,
],
```

После того как вы настроили опцию `tap` своего канала, вы готовы определить класс, который будет контролировать ваш экземпляр Monolog. Этому классу нужен только один метод: `__invoke`, который получает экземпляр `\Illuminate\Log\Logger`. Экземпляр `\Illuminate\Log\Logger` передает все вызовы методов базовому экземпляру Monolog:

```
<?php

namespace App\Logging;

use Illuminate\Log\Logger;
use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    /**
     * Настроить переданный экземпляр регистратора.
     *
     * @param \Illuminate\Log\Logger $logger
     * @return void
     */
    public function __invoke(Logger $logger): void
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(new LineFormatter(
                '[%datetime%] %channel%.%level_name%: %message% %context% %extra%'
            ));
        }
    }
}
```



Все ваши классы «тап» извлекаются через [контейнер служб](#), поэтому любые зависимости конструктора, которые им требуются, будут автоматически внедрены.

# Создание обработчика каналов Monolog

В Monolog есть множество [доступных обработчиков](#), а в Laravel из коробки не включены каналы для каждого из них. В некоторых случаях вам может потребоваться создать собственный канал, являющийся просто экземпляром определенного обработчика Monolog, у которого нет соответствующего драйвера журнала Laravel. Эти каналы могут быть легко созданы с помощью драйвера `monolog`.

При использовании драйвера `monolog` параметр конфигурации `handler` используется для указания того, какой обработчик будет создан. При желании любые параметры конструктора, необходимые обработчику, могут быть указаны с помощью опции конфигурации `with`:

```
'logentries' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\SyslogUdpHandler::class,
    'with' => [
        'host' => 'my.logentries.internal.datahubhost.company.com',
        'port' => '10000',
    ],
],
```

## Форматтеры Monolog

При использовании драйвера `monolog`, Monolog-класс `LineFormatter` будет использоваться как средство форматирования по умолчанию. Однако вы можете настроить тип средства форматирования, передаваемого обработчику, используя параметры конфигурации `formatter` и `formatter_with`:

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

Если вы используете обработчик Monolog, который может предоставлять свой собственный модуль форматирования, вы можете установить для параметра

конфигурации `formatter` значение `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
],
```

## Monolog Процессоры (Processors)

Monolog также может обрабатывать сообщения перед их записью в журнал. Вы можете создавать свои собственные процессоры или использовать [существующие процессоры, предлагаемые Monolog](#).

Если вы хотите кастомизировать процессоры для драйвера `monolog`, добавьте значение конфигурации `processors` в конфигурацию вашего канала:

```
'memory' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\StreamHandler::class,
    'with' => [
        'stream' => 'php://stderr',
    ],
    'processors' => [
        // Simple syntax...
        Monolog\Processor\MemoryUsageProcessor::class,
        // With options...
        [
            'processor' => Monolog\Processor\PsrLogMessageProcessor::class,
            'with' => ['removeUsedContextFields' => true],
        ],
    ],
],
```

## Создание каналов через фабрики

Если вы хотите определить полностью настраиваемый канал, в котором у вас есть полный контроль над созданием и конфигурацией Monolog, вы можете указать тип драйвера `custom` в файле конфигурации `config/logging.php`. Ваша конфигурация должна включать параметр `via`, содержащий имя класса фабрики, которая будет вызываться для создания экземпляра Monolog:

```
'channels' => [
    'example-custom-channel' => [
        'driver' => 'custom',
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

После того как вы настроили канал драйвера `custom`, вы готовы определить класс, который будет создавать ваш экземпляр Monolog. Этому классу нужен только один метод `__invoke`, который должен возвращать экземпляр регистратора Monolog. Метод получит массив конфигурации каналов в качестве единственного аргумента:

```
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Создать экземпляр собственного регистратора Monolog.
     *
     * @param array $config
     * @return \Monolog\Logger
     */
    public function __invoke(array $config): Logger
    {
        return new Logger(/* ... */);
    }
}
```

## # Просмотр сообщения журнала с помощью Pail

Часто вам может понадобиться просматривать журналы приложения в режиме реального времени. Например, при отладке проблемы или при мониторинге журналов приложения на предмет определенных типов ошибок.

Laravel Pail – это пакет, который позволяет вам легко погружаться в лог-файлы вашего приложения Laravel прямо из командной строки. В отличие от стандартной команды `tail`, Pail предназначен для работы с любым драйвером журналов,

включая Sentry или Flare. Кроме того, Pail предоставляет набор полезных фильтров, которые помогут вам быстро найти то, что вы ищете.

```
php artisan pail

→ php artisan pail

INFO Tailing application logs. Press Ctrl+C to exit
Use -v|-vv to show more details

13:31:54 INFO — This is some useful information.
                                         GET: / . Auth ID: guest

13:31:54 Exception — Error Processing Request
                                         GET: / . Auth ID: guest

13:31:54 WARNING — Something could be going wrong.
                                         GET: / . Auth ID: guest
```

## Установка

Laravel Pail поддерживает [PHP 8.2+](#) и расширение [PCNTL](#).

Чтобы начать работу, установите Pail в свой проект с помощью менеджера пакетов Composer:

```
composer require laravel/pail
```

## Использование

Чтобы начать отслеживать журналы, выполните команду `pail`:

```
php artisan pail
```

Чтобы увеличить детализацию вывода и избежать усечения (...), используйте опцию `-v`:

```
php artisan pail -v
```

Для максимальной детализации и отображения трассировок стека исключений используйте опцию `-vv`:

```
php artisan pail -vv
```

Чтобы прекратить отслеживание журналов, нажмите `Ctrl+C` в любой момент.

## Фильтрация логов

### --filter

Вы можете использовать опцию `--filter` для фильтрации журналов по их типу, файлу, сообщению и содержимому трассировки стека:

```
php artisan pail --filter="QueryException"
```

### --message

Чтобы фильтровать журналы только по их сообщениям, вы можете использовать опцию `--message`:

```
php artisan pail --message="User created"
```

### --level

Опцию `--level` можно использовать для фильтрации журналов по их [уровню](#):

```
php artisan tail --level=error
```

## --user

Чтобы отображать только те журналы, которые были записаны при аутентифицированным пользователем, вы можете указать идентификатор пользователя в опции `--user`:

```
php artisan tail --user=1
```

# Консоль Artisan

## # Введение

# Tinker (REPL)

## # Написание команд

# Генерация команд

# Структура команды

# Анонимные команды

# Изолированные команды

## # Определение вводимых данных

# Аргументы

# Параметры

# Массивы данных

# Описания вводимых данных

# Запрос отсутствующего ввода

## # Ввод/вывод команды

# Получение входных данных

# Запрос для ввода данных

# Вывод данных

## # Регистрация команд

## # Программное выполнение команд

# Вызов команд из других команд

## # Обработка сигналов

## # Настройка заготовок команд (stubs)

## # События

## # Введение

Artisan – это интерфейс командной строки, входящий в состав Laravel. Он предлагает ряд полезных команд, которые помогут при создании приложения.

Для просмотра списка всех доступных команд Artisan можно использовать команду `list`:

```
php artisan list
```

Каждая команда также включает в себя экран «справки», который отображает и описывает доступные аргументы и параметры команды. Чтобы просмотреть экран справки, используйте `help` перед именем команды:

```
php artisan help migrate
```

## Laravel Sail

Если вы используете [Laravel Sail](#) в качестве локальной среды разработки, не забудьте использовать командную строку `sail` для вызова команд Artisan. Sail выполнит ваши команды Artisan в контейнерах Docker вашего приложения:

```
./vendor/bin/sail artisan list
```

## Tinker (REPL)

Laravel Tinker – это мощный REPL для фреймворка Laravel, основанный на пакете [PsySH](#).

## Установка

Все приложения Laravel по умолчанию включают Tinker. Однако вы можете установить Tinker с помощью Composer, если вы ранее удалили его из своего приложения:

```
composer require laravel/tinker
```

Ищете графический интерфейс для взаимодействия с приложением Laravel? Зацените [Tinkerwell](#)!

## Использование

Tinker позволяет взаимодействовать полностью со всем приложением Laravel из командной строки, включая модели Eloquent, задачи, события и многое другое.

Чтобы войти в среду Tinker, выполните команду `tinker` Artisan:

```
php artisan tinker
```

Вы можете опубликовать конфигурационный файл Tinker с помощью команды `vendor:publish`:

```
php artisan vendor:publish --provider="Laravel\\Tinker\\TinkerServiceProvider"
```

Глобальный помощник `dispatch` и метод `dispatch` класса `Dispatchable` зависят от “garbage collection” для помещения задания в очередь. Следовательно, при использовании Tinker вы должны использовать `Bus::dispatch` или `Queue::push` для отправки заданий.

## Список разрешенных команд

Tinker использует список «разрешенных» команд, которые разрешено запускать Artisan в её среде. По умолчанию вы можете запускать команды `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `migrate:install`, `up` и `optimize`. Для добавления в этот список больше команд, добавьте их в массив `commands` конфигурационного файла `config/tinker.php`:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

## Черный список псевдонимов

Как правило, Tinker автоматически создает псевдонимы классов, когда вы взаимодействуете с ними в Tinker. Тем не менее вы можете запретить такое

поведение для некоторых классов, перечислив их в массиве `dont_alias` конфигурационного файла `config/tinker.php`:

```
'dont_alias' => [
    App\Models\User::class,
],
```

## # Написание команд

В дополнение к командам Artisan, вы можете создавать пользовательские команды. Команды обычно хранятся в каталоге `app\Console\Commands`; однако вы можете выбрать другое месторасположение, если эти команды могут быть загружены менеджером Composer.

## Генерация команд

Чтобы сгенерировать новую команду, используйте команду `make:command Artisan`. Эта команда поместит новый класс команды в каталог `app\Console\Commands` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его:

```
php artisan make:command SendEmails
```

## Структура команды

После создания команды следует заполнить свойства класса `$signature` и `$description`. Эти свойства будут отображаться на экране при использовании команды `list`. Свойство `$signature` также позволяет определять вводимые данные. Метод `handle` будет вызываться при выполнении команды. Вы можете разместить логику команды в этом методе.

Давайте рассмотрим пример команды. Обратите внимание, что мы можем запросить любые необходимые зависимости в методе `handle` команды. Контейнер служб Laravel автоматически внедрит все зависимости, типы которых объявлены в этом методе:

```
<?php
```

```
namespace App\Console\Commands;  
  
use App\Models\User;  
use App\Support\DripEmailer;  
use Illuminate\Console\Command;  
  
class SendEmails extends Command  
{  
    /**  
     * Имя и сигнатура консольной команды.  
     *  
     * @var string  
     */  
    protected $signature = 'mail:send {user}';  
  
    /**  
     * Описание консольной команды.  
     *  
     * @var string  
     */  
    protected $description = 'Send a marketing email to a user';  
  
    /**  
     * Выполнить консольную команду.  
     */  
    public function handle(DripEmailer $drip): void  
    {  
        $drip->send(User::find($this->argument('user')));  
    }  
}
```

Хорошей практикой повторного использования кода считается создание «простых» консольных команд с делегированием своих задач службам приложения. В приведенном примере мы внедряем класс службы для выполнения «затратной» отправки электронных писем.

## Коды завершения

Если из метода `handle` ничего не возвращается и команда выполняется успешно, команда завершится с кодом завершения `0`, что указывает на успех. Однако метод `handle` может дополнительно возвращать целое число, чтобы вручную указать код завершения команды:

```
$this->error('Something went wrong.');
```

```
return 1;
```

Если вы хотите «не выполнить» команду любым методом внутри команды, вы можете использовать метод `fail`. Метод `fail` немедленно прекратит выполнение команды и вернет код завершения `1`:

```
$this->fail('Something went wrong.');
```

## Анонимные команды

Анонимные команды обеспечивают альтернативу определению консольных команд в виде классов. Точно так же, как замыкания маршрутов являются альтернативой контроллерам.

Несмотря на то, что файл `routes/console.php` не определяет HTTP-маршруты, он определяет консольные точки входа (маршруты) в ваше приложение. В этом файле вы можете определить все консольные команды на основе замыканий, используя метод `Artisan::command`. Метод `command` принимает два аргумента: [сигнатура команды](#) и замыкание, которое получает аргументы и параметры команды:

```
Artisan::command('mail:send {user}', function (string $user) {
    $this->info("Sending email to: {$user}!");
});
```

Замыкание привязано к базовому экземпляру команды, поэтому у вас есть полный доступ ко всем вспомогательным методам, к которым вы обычно можете обращаться в команде, созданной с помощью класса.

## Типизация зависимостей

Помимо получения аргументов и параметров, замыкание анонимной команды также принимает дополнительные зависимости из [контейнера служб](#), необходимые для внедрения:

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, string $user) {
    $drip->send(User::find($user));
});
```

## Описания анонимных команд

При определении анонимных команд, можно использовать метод `purpose` для добавления описания команды. Это описание будет отображаться при запуске команд `php artisan list` и `php artisan help`:

```
Artisan::command('mail:send {user}', function (string $user) {
    // ...
})->purpose('Send a marketing email to a user');
```

## Изолированные команды

Для использования этой функции ваше приложение должно использовать `memcached`, `redis`, `dynamodb`, `database`, `file` или `array` в качестве кэш-драйвера по умолчанию. Кроме того, все серверы должны обмениваться данными с одним и тем же центральным сервером кэша.

Иногда вам может потребоваться, чтобы одновременно мог выполняться только один экземпляр команды. Для этого вы можете реализовать интерфейс [Illuminate\Contracts\Console\Isolatable](#) в вашем классе команды:

```
<?php
```

```
namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\Isolatable;

class SendEmails extends Command implements Isolatable
{
    // ...
}
```

Когда команда отмечена как `Isolatable`, Laravel автоматически добавит опцию `--isolated` к команде. Когда команда вызывается с этой опцией, Laravel гарантирует, что никакие другие экземпляры этой команды в данный момент не выполняются. Laravel достигает этого, пытаясь получить блокировку с использованием кэш-драйвера по умолчанию вашего приложения. Если другие экземпляры команды выполняются, команда не будет выполнена; однако команда все равно завершится с кодом успешного завершения:

```
php artisan mail:send 1 --isolated
```

Если вы хотите указать код статуса завершения, который команда должна вернуть, если она не может выполниться, вы можете предоставить его с помощью опции `isolated`:

```
php artisan mail:send 1 --isolated=12
```

## Идентификатор блокировки (Lock ID)

По умолчанию Laravel использует имя команды для генерации строкового ключа, который используется для получения блокировки в кэше вашего приложения. Однако вы можете настроить этот ключ, определив метод `isolatableId` в вашем классе команды Artisan, что позволяет вам интегрировать аргументы или опции команды в ключ:

```
/**
 * Get the isolatable ID for the command.
 */
public function isolatableId(): string
{
```

```
    return $this->argument('user');
}
```

## Срок действия блокировки

По умолчанию изоляционные блокировки истекают после завершения команды. Или, если команда прервана и не может быть завершена, срок действия блокировки истечет через один час. Однако вы можете настроить время действия блокировки, определив метод `isolationLockExpiresAt` в вашей команде:

```
use DateTimeInterface;
use DateInterval;

/**
 * Determine when an isolation lock expires for the command.
 */
public function isolationLockExpiresAt(): DateTimeInterface|DateInterval
{
    return now()->addMinutes(5);
}
```

## # Определение вводимых данных

При написании консольных команд обычно происходит сбор данных, получаемых от пользователя, с помощью аргументов или параметров. Laravel позволяет очень удобно определять входные данные, которые вы ожидаете от пользователя, используя свойство `$signature` команды. Свойство `$signature` позволяет определить имя, аргументы и параметры команды в едином выражительном синтаксисе, схожим с синтаксисом маршрутов.

## Аргументы

Все предоставленные пользователем аргументы и параметры заключаются в фигурные скобки. В следующем примере команда определяет один обязательный аргумент `user`:

```
/**
 * Имя и сигнатура консольной команды.
 *
 * @var string
```

```
 */
protected $signature = 'mail:send {user}';
```

По желанию можно сделать аргументы необязательными или определить значения по умолчанию:

```
// Необязательный аргумент ...
'mail:send {user?}'
```

```
// Необязательный аргумент с заданным по умолчанию значением ...
'mail:send {user=foo}'
```

## Параметры

Параметры, как и аргументы, являются разновидностью пользовательского ввода. Параметры должны иметь префикс в виде двух дефисов (`--`), при использовании их в командной строке. Существует два типа параметров: получающие значение, и те, которые его не получают. Параметры, которые не получают значение, служат логическими «переключателями». Давайте рассмотрим пример такого варианта:

```
/**
 * Имя и сигнатура консольной команды.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

В этом примере при вызове команды Artisan может быть указан переключатель `--queue`. Если переключатель `--queue` передан, то значение этого параметра будет `true`. В противном случае значение будет `false`:

```
php artisan mail:send 1 --queue
```

## Параметры со значениями

Давайте рассмотрим параметр, ожидающий значение. Если пользователь должен указать значение для параметра, то добавьте суффикс `=` к имени параметра:

```
/**  
 * Имя и сигнатура консольной команды.  
 *  
 * @var string  
 */  
protected $signature = 'mail:send {user} {--queue=}';
```

В этом примере пользователь может передать значение для параметра. Если параметр не указан при вызове команды, то его значение будет `null`:

```
php artisan mail:send 1 --queue=default
```

Параметру можно присвоить значение по умолчанию, указав его после имени. Если значение параметра не передано пользователем, то будет использовано значение по умолчанию:

```
'mail:send {user} {--queue=default}'
```

## Псевдонимы параметров

Чтобы назначить псевдоним при определении параметра, вы можете указать его перед именем параметра и использовать символ разделителя | для отделения псевдонима от полного имени параметра:

```
'mail:send {user} {--Q|queue}'
```

При вызове команды в терминале, псевдонимы параметров должны иметь префикс с одним дефисом, и символ = не должен использоваться при указании значения параметра:

```
php artisan mail:send 1 -Qdefault
```

## Массивы данных

Чтобы определить, что аргументы или параметры ожидают массив данных, используйте метасимвол \*. Во-первых, давайте рассмотрим пример, в котором

описывается аргумент как массив данных:

```
'mail:send {user*}'
```

При вызове этого метода аргументы `user` могут передаваться по порядку в командную строку. Например, следующая команда установит значение `user` как `1` и `2`:

```
php artisan mail:send 1 2
```

Метасимвол `*` можно комбинировать с необязательным определением аргумента, чтобы разрешить ноль или более экземпляров аргумента:

```
'mail:send {user?*}'
```

## Параметр со множеством значений

При определении параметра, ожидающего множество значений, каждое значение передаваемого команде параметра должно иметь префикс с именем параметра:

```
'mail:send {--id=*}'
```

Такую команду можно вызвать, передав несколько аргументов `--id`:

```
php artisan mail:send --id=1 --id=2
```

## Описания вводимых данных

Вы можете назначить описания входным аргументам и параметрам, отделив имя аргумента от описания с помощью двоеточия. Если вам нужно немного больше места для определения вашей команды, то распределите определение на несколько строк:

```
/**  
 * Имя и сигнатура консольной команды.
```

```
*  
* @var string  
*/  
protected $signature = 'mail:send  
    {user : The ID of the user}  
    {--queue : Whether the job should be queued}';
```

## Запрос отсутствующего ввода

Если ваша команда содержит обязательные аргументы, пользователь получит сообщение об ошибке, если они не были предоставлены. В качестве альтернативы, вы можете настроить вашу команду так, чтобы автоматически запрашивать пользователя при отсутствии необходимых аргументов, реализовав интерфейс [PromptsForMissingInput](#):

```
<?php  
  
namespace App\Console\Commands;  
  
use Illuminate\Console\Command;  
use Illuminate\Contracts\Console\PromptsForMissingInput;  
  
class SendEmails extends Command implements PromptsForMissingInput  
{  
    /**  
     * Имя и сигнатура консольной команды.  
     *  
     * @var string  
     */  
    protected $signature = 'mail:send {user}';  
  
    // ...  
}
```

Если Laravel должен получить обязательный аргумент от пользователя, он автоматически запросит у пользователя этот аргумент, формулируя вопрос разумно с использованием имени или описания аргумента. Если вы хотите настроить вопрос, используемый для получения обязательного аргумента, реализуйте метод [promptForMissingArgumentsUsing](#), возвращающий массив вопросов с ключами соответствующими именам аргументов:

```
/**  
 * Prompt for missing input arguments using the returned questions.
```

```
* @return array<string, string>
*/
protected function promptForMissingArguments(): array
{
    return [
        'user' => 'Which user ID should receive the mail?',
    ];
}
```

Вы также можете указать текст заполнителя, используя кортеж, содержащий вопрос и заполнитель:

```
return [
    'user' => ['Which user ID should receive the mail?', 'E.g. 123'],
];
```

Если вы хотите полностью контролировать запрос, вы можете предоставить замыкание, которое будет запрашивать пользователя и возвращать его ответ:

```
use App\Models\User;
use function Laravel\Prompts\search;

// ...

return [
    'user' => fn () => search(
        label: 'Search for a user:',
        placeholder: 'E.g. Taylor Otwell',
        options: fn ($value) => strlen($value) > 0
            ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
            : []
    ),
];
```



Подробная документация по [Laravel Prompts](#) содержит дополнительную информацию о доступных запросах и их использовании.

Если вы хотите запросить у пользователя выбор или ввод опций, вы можете включить подсказки в метод `handle` вашей команды. Однако, если вы хотите запрашивать у пользователя только тогда, когда ему было автоматически предложено ввести отсутствующие аргументы, вы можете реализовать метод `afterPromptingForMissingArguments`:

```
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use function Laravel\Prompts\confirm;

// ...

/**
 * Выполнить действия после запроса пользователя относительно отсутствующих аргументов
 */
protected function afterPromptingForMissingArguments(InputInterface $input, OutputInterface $output)
{
    $input->setOption('queue', confirm(
        label: 'Would you like to queue the mail?',
        default: $this->option('queue')
    ));
}
```

## # Ввод/вывод команды

### Получение входных данных

Во время выполнения команды вам, вероятно, потребуется получить доступ к значениям аргументов и параметров, принятых командой. Для этого вы можете использовать методы `argument` и `option`. Если аргумент или параметр не существует, то будет возвращено значение `null`.

```
/**
 * Выполнить консольную команду.
 */
public function handle(): void
{
    $userId = $this->argument('user');

    //
}
```

Если вам нужно получить все аргументы в виде массива, вызовите метод `arguments`:

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

Параметры могут быть получены так же легко, как и аргументы, используя метод `option`. Чтобы получить все параметры в виде массива, вызовите метод `options`:

```
// Получение определенного параметра ...
$queueName = $this->option('queue');

// Получение всех параметров в виде массива ...
$options = $this->options();
```

## Запрос для ввода данных

[Laravel Prompts](#) это PHP-пакет для добавления красивых и удобных форм в ваше консольное приложение с функциями, аналогичными браузеру, включая текст заполнителя и проверку данных.

Помимо отображения вывода, вы можете попросить пользователя предоставить данные во время выполнения вашей команды. Метод `ask` отобразит пользователю указанный вопрос, примет его ввод, а затем вернет эти данные, полученные от пользователя, обратно в команду:

```
/**
 * Выполнить консольную команду.
 */
public function handle(): void
{
    $name = $this->ask('What is your name?');

    // ...
}
```

Метод `ask` также принимает необязательный второй аргумент, который определяет значение по умолчанию, возвращаемое, если пользователь не предоставил ввод:

```
$name = $this->ask('What is your name?', 'Taylor');
```

Метод `secret` похож на `ask`, но ввод пользователя не будет виден ему в консоли при вводе. Этот метод полезен при запросе конфиденциальной информации, например, пароля:

```
$password = $this->secret('What is the password?');
```

## Запрос подтверждения

Если вам нужно получить от пользователя простое подтверждение «yes or no», то вы можете использовать метод `confirm`. По умолчанию этот метод возвращает значение `false`. Однако, если пользователь вводит `y` или `yes` в ответ на запрос, то метод возвращает `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    // ...
}
```

По желанию можно указать, что запрос подтверждения должен по умолчанию возвращать `true`, передав `true` в качестве второго аргумента метода `confirm`:

```
if ($this->confirm('Do you wish to continue?', true)) {
    // ...
}
```

## Автозавершение

Метод `anticipate` используется для автоматического завершения возможных вариантов. Пользователь по-прежнему может дать любой ответ, независимо от подсказок автозавершения:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

В качестве альтернативы, вы можете передать замыкание в качестве второго аргумента метода `anticipate`. Замыкание будет вызываться каждый раз, когда пользователь вводит символ. Замыкание должно принимать строковый параметр, содержащий введенные пользователем данные, и возвращать массив вариантов для автозавершения:

```
$name = $this->anticipate('What is your address?', function (string $input) {  
    // Вернуть варианты для автоматического завершения ...  
});
```

## Вопросы с множественным выбором

Если нужно предоставить пользователю предопределенный набор вариантов для выбора при задании вопроса, то используйте метод `choice`. Вы можете установить индекс массива для возвращаемого по умолчанию значения, если не выбран ни один из вариантов, передав индекс в качестве третьего аргумента метода:

```
$name = $this->choice(  
    'What is your name?',  
    ['Taylor', 'Dayle'],  
    $defaultIndex  
);
```

Кроме того, метод `choice` принимает необязательные четвертый и пятый аргументы для определения максимального количества попыток выбора действительного ответа и того, разрешен ли множественный выбор:

```
$name = $this->choice(  
    'What is your name?',  
    ['Taylor', 'Dayle'],  
    $defaultIndex,  
    $maxAttempts = null,  
    $allowMultipleSelections = false  
);
```

## Вывод данных

Чтобы вывести в консоль, используйте методы `line`, `info`, `comment`, `question`, `warn` и `error`. Каждый из этих методов будет использовать соответствующие ANSI-цвета.

Например, давайте покажем пользователю некоторую общую информацию.

Обычно метод `info` отображается в консоли в виде зеленого текста:

```
/**  
 * Выполнить консольную команду.  
 */  
public function handle(): void  
{  
    // ...  
  
    $this->info('The command was successful!');  
}
```

Для отображения сообщения об ошибке используйте метод `error`. Текст сообщения об ошибке обычно отображается красным цветом:

```
$this->error('Something went wrong!');
```

Вы можете использовать метод `line` для отображения простого неокрашенного текста:

```
$this->line('Display this on the screen');
```

Вы можете использовать метод `newLine` для отображения пустой строки:

```
// Вывести одну пустую строку ...  
$this->newLine();  
  
// Вывести три пустые строки ...  
$this->newLine(3);
```

## Таблицы

Метод `table` упрощает корректное форматирование нескольких строк / столбцов данных. Все, что вам нужно сделать, это указать имена столбцов и данные для таблицы, и Laravel автоматически рассчитает подходящую ширину и высоту таблицы:

```
use App\Models\User;

$this->table(
    [ 'Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

## Индикаторы выполнения

Для длительно выполняемых задач было бы полезно показать индикатор выполнения, информирующий пользователя о том, насколько завершена задача. Используя метод `withProgressBar`, Laravel будет отображать индикатор выполнения и продвигать его для каждой итерации на заданное повторяемое значение:

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function (User $user) {
    $this->performTask($user);
});
```

Иногда может потребоваться больший контроль над продвижением индикатора выполнения. Сначала определите общее количество шагов, через которые будет проходить процесс. Затем продвигайте индикатор выполнения после обработки каждого элемента:

```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

Для получения дополнительной информации  
ознакомьтесь с [разделом документации компонента](#)  
[Symfony Progress Bar](#).

## # Регистрация команд

По умолчанию Laravel автоматически регистрирует все команды в каталоге `app/Console/Commands`. Однако вы можете поручить Laravel сканировать другие каталоги на наличие команд Artisan, используя метод `withCommands` в файле `bootstrap/app.php` вашего приложения:

```
->withCommands([
    __DIR__.'/../app/Domain/Orders/Commands',
])
```

При необходимости вы также можете зарегистрировать команды вручную, указав имя класса команды в методе `withCommands`:

```
use App\Domain\Orders\Commands\SendEmails;

->withCommands([
    SendEmails::class,
])
```

Когда Artisan загрузится, все команды в вашем приложении будут обработаны [сервисным контейнером](#) и зарегистрированы в Artisan.

## # Программное выполнение команд

По желанию можно выполнить команду Artisan за пределами CLI. Например, вы можете запустить команду Artisan в маршруте или контроллере. Для этого можно использовать метод `call` фасада `Artisan`. Метод `call` принимает в качестве первого аргумента либо имя сигнатуры команды, либо имя класса, а в качестве второго – массив параметров команды. Будет возвращен код выхода / возврата:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);
    //...
});
```

Кроме того, вы можете передать методу `call` команду полностью в виде строки:

```
Artisan::call('mail:send 1 --queue=default');
```

## Передача массива значений

Если ваша команда определяет параметр, который принимает массив, то вы можете передать массив значений этому параметру:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

## Передача значений логического типа

Если необходимо указать значение параметра, который не принимает строковые значения, например флаг `--force` в команде `migrate:refresh`, то вы должны передать `true` или `false` как значение параметра:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

## Очереди команд Artisan

Используя метод `queue` фасада `Artisan`, вы можете даже поставить команды Artisan в очередь, чтобы они обрабатывались в фоновом режиме [обработчиком очереди](#). Перед использованием этого метода убедитесь, что вы настроили очереди и был запущен слушатель очереди:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    //...
});
```

Используя методы `onConnection` и `onQueue`, вы также можете указать соединение или очередь, в которую должна быть отправлена команда Artisan:

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

## Вызов команд из других команд

По желанию можно вызвать другие команды из существующей команды Artisan. Вы можете сделать это с помощью метода `call`. Метод `call` принимает имя команды и массив аргументов / параметров команды:

```
/**
 * Выполнить консольную команду.
 */
public function handle(): void
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //...
}
```

Если вы хотите вызвать другую консольную команду в тихом режиме, то используйте метод `callSilently`. Метод `callSilently` имеет ту же сигнатуру, что и метод `call`:

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

## # Обработка сигналов

Как вы, возможно, знаете, операционные системы позволяют отправлять сигналы запущенным процессам. Например, сигнал `SIGTERM` используется операционными системами для запроса программе о завершении выполнения. Если вы хотите прослушивать сигналы в ваших консольных командах Artisan и выполнять код при их возникновении, вы можете использовать метод `trap`:

```
/**
 * Выполнить консольную команду.
 */
public function handle(): void
{
    $this->trap(SIGTERM, fn () => $this->shouldKeepRunning = false);

    while ($this->shouldKeepRunning) {
        // ...
    }
}
```

Для прослушивания нескольких сигналов сразу, вы можете предоставить массив сигналов методу `trap`:

```
$this->trap([SIGTERM, SIGQUIT], function (int $signal) {
    $this->shouldKeepRunning = false;

    dump($signal); // SIGTERM / SIGQUIT
});
```

## # Настройка заготовок команд (stubs)

Команды `make` консоли Artisan используются для создания различных классов, таких как контроллеры, задания, миграции и тесты. Эти классы создаются с помощью файлов «заготовок», которые заполняются значениями на основе ваших входных данных. Однако, иногда может потребоваться внести небольшие изменения в файлы, создаваемые с помощью Artisan. Для этого можно использовать команду `stub:publish`, чтобы опубликовать наиболее распространенные заготовки для их дальнейшего изменения:

```
php artisan stub:publish
```

Опубликованные заготовки будут расположены в каталоге `stubs` корня вашего приложения. Любые изменения, внесенные вами в эти заготовки, будут учтены при создании соответствующих классов с помощью команд `make` Artisan.

## # События

Artisan запускает три события при выполнении команд:

`Illuminate\Console\Events\ArtisanStarting`, `Illuminate\Console\Events\CommandStarting`, и `Illuminate\Console\Events\CommandFinished`. Событие `ArtisanStarting` выполняется сразу после запуска Artisan. Затем событие `CommandStarting` выполняется непосредственно перед запуском команды. Наконец, событие `CommandFinished` выполняется после завершения команды.

# Трансляция (broadcast) событий

## # Введение

### # Установка на стороне сервера

# Конфигурирование

# Reverb

# Pusher Channels

# Ably

### # Установка на стороне клиента

# Reverb

# Pusher Channels

# Ably

## # Обзор концепции

# Пример использования

## # Определение транслируемых событий

# Имя транслируемого события

# Данные трансляции

# Очередь трансляции

# Условия трансляции

## # Авторизация каналов

# Определение авторизации канала

# Определение класса канала

## # Трансляция событий

# Трансляция событий только остальным пользователям

# Настройка подключения

# Анонимные события

## # Прием трансляций

# Прослушивание событий

# Покидание канала

# Пространства имён

## # Каналы присутствия

- # Авторизация каналов присутствия
- # Присоединение к каналам присутствия
- # Трансляция на каналы присутствия

# **Трансляция моделей**

- # Соглашение о трансляции моделей
- # Прослушивание трансляций моделей

# **Клиентские события**

# **Уведомления**

## # Введение

Во многих современных веб-приложениях веб-сокеты используются для реализации пользовательских интерфейсов, обновляемых в реальном времени. Когда некоторые данные обновляются на сервере, тогда обычно отправляется сообщение через соединение WebSocket для обработки клиентом. Веб-сокеты предоставляют более эффективную альтернативу постоянному опросу сервера вашего приложения на предмет изменений данных, которые должны быть отражены в вашем пользовательском интерфейсе.

Например, представьте, что ваше приложение может экспортировать данные пользователя в файл CSV и отправлять этот файл ему по электронной почте. Однако создание этого CSV-файла занимает несколько минут, поэтому вы можете создать и отправить CSV-файл по почте, поместив [задание в очередь](#). Когда файл CSV будет создан и отправлен пользователю, тогда мы можем использовать **широковещание** для отправки события `App\Events\UserDataExported`, которое будет получено в JavaScript нашего приложения. Как только событие будет получено, мы можем отобразить сообщение пользователю о том, что его файл CSV был отправлен ему по электронной почте без необходимости в обновлении страницы.

Чтобы помочь вам в создании подобного рода функционала, Laravel упрощает «вещание» серверных [событий](#) Laravel через соединение WebSocket. Трансляция ваших событий Laravel позволяет вам использовать одни и те же имена событий и данные между серверным приложением Laravel и клиентским JavaScript-приложением.

Основные концепции широковещательной передачи просты: клиенты подключаются к именованным каналам во внешнем интерфейсе, в то время как ваше приложение Laravel транслирует события на эти каналы во внутреннем

интерфейсе. Эти события могут содержать любые дополнительные данные, которые вы хотите сделать доступными для внешнего интерфейса.

## Поддерживаемые драйверы

По умолчанию Laravel содержит три серверных драйвера трансляции на выбор: [Laravel Reverb](#), [Pusher Channels](#), и [Ably](#)

Прежде чем ближе ознакомиться с трансляцией событий, убедитесь, что вы прочитали документацию Laravel о [событиях и слушателях](#).

## # Установка на стороне сервера

Чтобы начать использовать трансляцию событий Laravel, нам нужно выполнить некоторую настройку в приложении Laravel, а также установить некоторые пакеты.

Трансляция событий осуществляется серверным драйвером трансляции, который транслирует ваши события Laravel, получаемые браузером клиента через Laravel Echo (библиотека JavaScript). Не волнуйтесь – мы рассмотрим каждую часть процесса установки шаг за шагом.

## Конфигурирование

Вся конфигурация трансляций событий вашего приложения хранится в конфигурационном файле `config/broadcasting.php`. Не волнуйтесь, если этот каталог не существует в вашем приложении; он будет создан при запуске Artisan-команды `install:broadcasting`.

Laravel из коробки поддерживает несколько драйверов трансляции: [Laravel Reverb](#), [Pusher Channels](#), [Ably](#), а также драйвер `log` для локальной разработки и отладки. Кроме того, поддерживается драйвер `null`, который позволяет полностью отключить трансляцию во время тестирования. В конфигурационном файле `config/broadcasting.php` содержится пример конфигурации для каждого из этих драйверов.

## Установка

По умолчанию трансляция не включена в новых приложениях Laravel. Вы можете включить трансляцию с помощью Artisan-команды `install:broadcasting`:

```
php artisan install:broadcasting
```

Команда `install:broadcasting` создаст файл конфигурации `config/broadcasting.php`.

Кроме того, команда создаст файл `routes/channels.php`, в котором вы можете зарегистрировать маршруты авторизации трансляции и обратные вызовы вашего приложения.

## Конфигурирование очереди

Прежде чем транслировать какие-либо события, вам следует сначала настроить и запустить [обработчик очереди](#). Вся трансляция событий выполняются через задания в очереди, поэтому транслируемые события не оказывают серьезного влияния на время отклика вашего приложения.

## Reverb

При запуске команды `install:broadcasting` вам будет предложено установить [Laravel Reverb](#). Конечно, вы также можете установить Reverb вручную, используя менеджер пакетов Composer.

```
composer require laravel/reverb
```

После установки пакета вы можете запустить команду установки Reverb, чтобы опубликовать конфигурацию, добавить необходимые переменные среды Reverb и включить трансляцию событий в вашем приложении:

```
php artisan reverb:install
```

Подробные инструкции по установке и использованию Reverb можно найти в [документации Reverb](#).

# Pusher Channels

Если вы планируете транслировать свои события с помощью [Pusher Channels](#), то вам следует установить PHP SDK Pusher Channels с помощью менеджера пакетов Composer:

```
composer require pusher/pusher-php-server
```

Далее, вы должны настроить свои учетные данные Pusher Channels в конфигурационном файле [config/broadcasting.php](#). Пример конфигурации Pusher Channels уже содержится в этом файле, что позволяет быстро указать параметры `key`, `secret`, и `app_id`. Обычно вам следует настроить учетные данные Pusher Channels в файле [.env](#) вашего приложения:

```
PUSHER_APP_ID="your-pusher-app-id"
PUSHER_APP_KEY="your-pusher-key"
PUSHER_APP_SECRET="your-pusher-secret"
PUSHER_HOST=
PUSHER_PORT=443
PUSHER_SCHEME="https"
PUSHER_APP_CLUSTER="mt1"
```

Конфигурация `pusher` в файле [config/broadcasting.php](#) также позволяет вам указывать дополнительные параметры, которые поддерживаются Pusher, например, `cluster`.

Затем установите для переменной среды `BROADCAST_CONNECTION` значение `pusher` в файле [.env](#) вашего приложения:

```
BROADCAST_CONNECTION=pusher
```

И, наконец, вы готовы установить и настроить [Laravel Echo](#), который будет получать транслируемые события на клиентской стороне.

**Ably**

Ниже приведено описание того, как использовать Ably в режиме “совместимости с Pusher”. Однако команда Ably рекомендует и поддерживает вещатель и клиент Echo, способные использовать уникальные возможности, предлагаемые Ably. Для получения дополнительной информации о использовании поддерживаемых Ably драйверов обратитесь к [документации Ably по Laravel broadcaster](#).

Если вы планируете транслировать свои события с помощью [Ably](#), то вам следует установить PHP SDK Ably с помощью менеджера пакетов Composer:

```
composer require ably/ably-php
```

Далее, вы должны настроить свои учетные данные Ably в конфигурационном файле [config/broadcasting.php](#). Пример конфигурации Ably уже содержится в этом файле, что позволяет быстро указать параметр `key`. Как правило, это значение должно быть установлено через [переменную окружения ABLY\\_KEY](#):

```
ABLY_KEY=your-ably-key
```

Затем установите для переменной среды `BROADCAST_CONNECTION` значение `ably` в файле `.env` вашего приложения:

```
BROADCAST_CONNECTION=ably
```

И, наконец, вы готовы установить и настроить [Laravel Echo](#), который будет получать транслируемые события на клиентской стороне.

## # Установка на стороне клиента

Reverb

[Laravel Echo](#) — это библиотека JavaScript, которая позволяет без труда подписываться на каналы и прослушивать события, транслируемые вашим серверным драйвером вещания. Вы можете установить Echo через менеджер пакетов NPM. В этом примере мы также установим пакет [pusher-js](#), поскольку Reverb использует протокол Pusher для подисок, каналов и сообщений WebSocket:

```
npm install --save-dev laravel-echo pusher-js
```

После установки Echo вы готовы создать новый экземпляр Echo в JavaScript вашего приложения. Отличное место для этого — внизу файла [resources/js/bootstrap.js](#), который входит в состав фреймворка Laravel. По умолчанию в этот файл уже включен пример конфигурации Echo — вам просто нужно раскомментировать его и обновить параметр конфигурации `broadcaster` на `reverb`:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'reverb',
    key: import.meta.env.VITE_REVERB_APP_KEY,
    wsHost: import.meta.env.VITE_REVERB_HOST,
    wsPort: import.meta.env.VITE_REVERB_PORT,
    wssPort: import.meta.env.VITE_REVERB_PORT,
    forceTLS: (import.meta.env.VITE_REVERB_SCHEME ?? 'https') === 'https',
    enabledTransports: ['ws', 'wss'],
});

});
```

Далее вам следует скомпилировать ресурсы вашего приложения:

```
npm run build
```

Для трансляции Laravel Echo `reverb` требуется laravel-echo v1.16.0+.

# Pusher Channels

[Laravel Echo](#) — это JavaScript-библиотека, которая упрощает подписку на каналы и прослушивание событий, транслируемые вашим серверным драйвером трансляции. Echo также использует пакет NPM `pusher-js` для реализации протокола Pusher для подписок, каналов и сообщений WebSocket.

Команда Artisan `install:broadcasting` автоматически устанавливает для вас пакеты `laravel-echo` и `pusher-js`; однако вы также можете установить эти пакеты вручную через NPM:

```
npm install --save-dev laravel-echo pusher-js
```

После установки Echo вы готовы создать новый экземпляр Echo в JavaScript вашего приложения. Команда `install:broadcasting` создает файл конфигурации Echo по адресу `resources/js/echo.js`; однако конфигурация по умолчанию в этом файле предназначена для Laravel Reverb. Вы можете скопировать конфигурацию ниже, чтобы перенести вашу конфигурацию на Pusher:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

Далее вам следует определить соответствующие значения для переменных среды Pusher в файле `.env` вашего приложения. Если эти переменные еще не существуют в вашем файле `.env`, вам следует добавить их:

```
PUSHER_APP_ID="your-pusher-app-id"
PUSHER_APP_KEY="your-pusher-key"
PUSHER_APP_SECRET="your-pusher-secret"
PUSHER_HOST=
PUSHER_PORT=443
PUSHER_SCHEME="https"
```

```
PUSHER_APP_CLUSTER="mt1"

VITE_APP_NAME="${APP_NAME}"
VITE_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
VITE_PUSHER_HOST="${PUSHER_HOST}"
VITE_PUSHER_PORT="${PUSHER_PORT}"
VITE_PUSHER_SCHEME="${PUSHER_SCHEME}"
VITE_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

После того, как вы настроили конфигурацию Echo в соответствии с потребностями вашего приложения, вы можете скомпилировать ресурсы вашего приложения:

```
npm run build
```

Чтобы узнать больше о компиляции JavaScript-исходников вашего приложения, обратитесь к документации [Vite](#).

## Использование существующего экземпляра клиента

Если у вас уже есть предварительно настроенный экземпляр клиента Pusher Channels, который вы бы хотели использовать в Echo, то вы можете передать его Echo с помощью свойства конфигурации `client`:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key'
}

window.Echo = new Echo({
    ...options,
    client: new Pusher(options.key, options)
});
```

Ниже приведено описание того, как использовать Ably в режиме “совместимости с Pusher”. Однако команда Ably рекомендует и поддерживает вещатель и клиент Echo, способные использовать уникальные возможности, предлагаемые Ably. Для получения дополнительной информации о использовании поддерживаемых Ably драйверов обратитесь к [документации Ably по Laravel broadcaster](#).

[Laravel Echo](#) — это JavaScript-библиотека, которая позволяет без труда подписываться на каналы и прослушивать события, транслируемые вашим серверным драйвером трансляции. Echo также использует пакет NPM `pusher-js` для реализации протокола Pusher для подписок, каналов и сообщений WebSocket.

Команда Artisan `install:broadcasting` автоматически устанавливает для вас пакеты `laravel-echo` и `pusher-js`; однако вы также можете установить эти пакеты вручную через NPM:

```
npm install --save-dev laravel-echo pusher-js
```

**Прежде чем продолжить, вы должны включить поддержку протокола Pusher в настройках вашего приложения Ably. Вы можете включить эту функцию в разделе настроек «Protocol Adapter Settings» панели вашего приложения Ably.**

После установки Echo вы готовы создать новый экземпляр Echo в JavaScript вашего приложения. Команда `install:broadcasting` создает файл конфигурации Echo по адресу `resources/js/echo.js`; однако конфигурация по умолчанию в этом файле предназначена для Laravel Reverb. Вы можете скопировать конфигурацию ниже, чтобы перенести ее в Ably:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
```

```
broadcaster: 'pusher',
key: import.meta.env.VITE_ABLY_PUBLIC_KEY,
wsHost: 'realtime-pusher.ably.io',
wsPort: 443,
disableStats: true,
encrypted: true,
});
```

Возможно, вы заметили, что наша конфигурация Echo для Ably ссылается на переменную окружения `VITE_ABLY_PUBLIC_KEY`. Значение этой переменной должно быть вашим публичным ключом Ably. Ваш публичный ключ – это часть ключа Ably перед символом `:`.

После того как вы настроили конфигурацию Echo в соответствии с вашими потребностями, вы можете скомпилировать исходники вашего приложения:

```
npm run dev
```

Чтобы узнать больше о компиляции JavaScript-исходников вашего приложения, обратитесь к документации [Vite](#).

## # Обзор концепции

Трансляция событий Laravel позволяет транслировать серверные события Laravel в JavaScript-приложение на клиентской стороне, используя драйверный подход к WebSockets. В настоящее время Laravel поставляется с драйверами [Pusher Channels](#) и [Ably](#). События могут быть легко обработаны на стороне клиента с помощью JavaScript-пакета [Laravel Echo](#).

События транслируются по «каналам», которые могут быть публичными или частными. Любой посетитель вашего приложения может подписаться на публичный канал без какой-либо аутентификации или авторизации; однако, чтобы подписаться на частный канал, пользователь должен быть аутентифицирован и авторизован для прослушивания событий на этом канале.

# Пример использования

Прежде чем углубляться в каждый аспект трансляции событий, давайте сделаем общий обзор на примере интернет-магазина.

Предположим, что в нашем приложении у нас есть страница, которая позволяет пользователям просматривать статус доставки своих заказов. Предположим также, что событие `OrderShipmentStatusUpdated` запускается, когда приложение обрабатывает обновление статуса доставки:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

## Интерфейс `ShouldBroadcast`

Когда пользователь просматривает один из своих заказов, мы не хотим, чтобы ему приходилось обновлять страницу для просмотра статуса обновлений. Вместо этого мы хотим транслировать обновления в приложение по мере их создания. Итак, нам нужно пометить событие `OrderShipmentStatusUpdated` интерфейсом `ShouldBroadcast`.

Это проинструментирует Laravel транслировать событие при его запуске:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * Экземпляр заказа.
     *
     * @var \App\Models\Order
     */
    public $order;
}
```

Интерфейс `ShouldBroadcast` требует, чтобы в нашем классе события был определен метод `broadcastOn`. Этот метод отвечает за возврат каналов, по которым должно транслироваться событие. Пустая заглушка этого метода уже определена в сгенерированных классах событий, поэтому нам нужно только заполнить ее реализацию. Мы хотим, чтобы только создатель заказа мог просматривать статус обновления, поэтому мы будем транслировать событие на частном канале, привязанном к конкретному заказу:

```
use Illuminate\\Broadcasting\\Channel;
use Illuminate\\Broadcasting\\PrivateChannel;

/**
 * Получить каналы трансляции события.
 */
public function broadcastOn(): Channel
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

Если вы хотите, чтобы событие передавалось по нескольким каналам, вы можете вернуть вместо этого `array`:

```
use Illuminate\\Broadcasting\\PrivateChannel;

/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\\Broadcasting\\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PrivateChannel('orders.'.$this->order->id),
        // ...
    ];
}
```

## Авторизация каналов

Помните, что пользователи должны иметь разрешение на прослушивание частных каналов. Мы можем определить наши правила авторизации каналов в файле `routes/channels.php` нашего приложения. В этом примере нам нужно убедиться, что

любой пользователь, пытающийся прослушивать частный канал `orders.1`, на самом деле является создателем заказа:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

Метод `channel` принимает два аргумента: имя канала и замыкание, которое возвращает `true` или `false`, указывая тем самым, имеет ли пользователь право прослушивать канал.

Все замыкания авторизации получают текущего аутентифицированного пользователя в качестве своего первого аргумента и любые дополнительные параметры в качестве своих последующих аргументов. В этом примере мы используем заполнитель `{orderId}`, чтобы указать, что часть «ID» имени канала является параметром.

## Прослушивание трансляций событий

Далее все, что остается, – это прослушивать событие в нашем JavaScript-приложении. Мы можем сделать это с помощью [Laravel Echo](#). Во-первых, мы будем использовать метод `private` для подписки на частный канал. Затем мы можем использовать метод `listen` для прослушивания события `OrderShipmentStatusUpdated`. По умолчанию все публичные свойства события будут включены в трансляцию события:

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
   });
```

## # Определение транслируемых событий

Чтобы сообщить Laravel, что какое-то событие должно транслироваться, вы должны реализовать интерфейс [Illuminate\Contracts\Broadcasting\ShouldBroadcast](#) в классе события. Этот интерфейс уже импортирован во все классы событий,

сгенерированные фреймворком, поэтому вы с легкостью можете добавить его к любому из ваших событий.

Интерфейс `ShouldBroadcast` требует, чтобы вы реализовали единственный метод: `broadcastOn`. Метод `broadcastOn` должен возвращать канал или массив каналов, по которым должно транслироваться событие. Каналы должны быть экземплярами `Channel`, `PrivateChannel` или `PresenceChannel`. Экземпляры `Channel` представляют собой публичные каналы, на которые может подписаться любой пользователь, в то время как `PrivateChannels` и `PresenceChannels` представляют собой частные каналы, для которых требуется [авторизация канала](#):

```
<?php
```

```
namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * Создать новый экземпляр события.
     */
    public function __construct(
        public User $user,
    ) {}

    /**
     * Получить каналы трансляции события.
     *
     * @return array<int, \Illuminate\Broadcasting\Channel>
     */
    public function broadcastOn(): array
    {
        return [
            new PrivateChannel('user.'.$this->user->id),
        ];
    }
}
```

После реализации интерфейса `ShouldBroadcast` вам нужно только запустить событие, как обычно. После того как событие будет запущено, задание в очереди автоматически транслирует событие, используя указанный вами драйвер трансляции.

## Имя транслируемого события

По умолчанию Laravel будет транслировать событие, используя имя класса события. Однако вы можете изменить имя транслируемого события, определив для события метод `broadcastAs`:

```
/**  
 * Имя транслируемого события.  
 */  
public function broadcastAs(): string  
{  
    return 'server.created';  
}
```

Если вы измените имя транслируемого события с помощью метода `broadcastAs`, то вы должны убедиться, что зарегистрировали ваш слушатель с ведущим символом `.`. Это проинструктирует Echo не добавлять пространство имен приложения к событию:

```
.listen('.server.created', function (e) {  
    ....  
});
```

## Данные трансляции

При трансляции события, все его публичные свойства автоматически сериализуются и транслируются как полезная нагрузка события, что позволяет вам получить доступ к любым его публичным данным из вашего JavaScript-приложения. Так, например, если ваше событие имеет единственное публичное свойство `$user`, представляющее собой модель Eloquent, то полезная нагрузка при трансляции события будет:

```
{  
    "user": {
```

```
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

Однако, если вы хотите иметь более точный контроль над полезной нагрузкой трансляции, то вы можете определить метод `broadcastWith` вашего события. Этот метод должен возвращать массив данных, которые вы хотите использовать в качестве полезной нагрузки при трансляции события:

```
/**
 * Получите данные для трансляции.
 *
 * @return array<string, mixed>
 */
public function broadcastWith(): array
{
    return ['id' => $this->user->id];
}
```

## Очередь трансляции

По умолчанию каждое транслируемое событие помещается в очередь по умолчанию и соединение очереди по умолчанию, указанные в вашем конфигурационном файле `config/queue.php`. Вы можете изменить соединение очереди и имя, используемое вещателем, определив свойства `connection` и `queue` в вашем классе события:

```
/**
 * Имя соединения очереди, которое будет использоваться при трансляции события.
 *
 * @var string
 */
public $connection = 'redis';

/**
 * Имя очереди, в которую нужно поместить задание трансляции.
 *
 * @var string
 */
public $queue = 'default';
```

В качестве альтернативы вы можете настроить имя очереди, определив в методе `broadcastQueue` вашего события:

```
/**  
 * The name of the queue on which to place the broadcasting job.  
 */  
public function broadcastQueue(): string  
{  
    return 'default';  
}
```

Если вы хотите транслировать свое событие с помощью очереди `sync` вместо драйвера очереди по умолчанию, то вы можете реализовать интерфейс `ShouldBroadcastNow` вместо `ShouldBroadcast`:

```
<?php  
  
use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;  
  
class OrderShipmentStatusUpdated implements ShouldBroadcastNow  
{  
    // ...  
}
```

## Условия трансляции

Иногда необходимо транслировать событие только в том случае, если выполняется определенное условие. Вы можете определить эти условия, добавив метод `broadcastWhen` в ваш класс события:

```
/**  
 * Определить, условия трансляции события.  
 */  
public function broadcastWhen(): bool  
{  
    return $this->order->value > 100;  
}
```

## Трансляция и транзакции базы данных

Когда транслируемые события отправляются в транзакциях базы данных, они могут быть обработаны очередью до того, как транзакция базы данных будет зафиксирована. Когда это происходит, любые обновления, внесенные вами в модели или записи базы данных во время транзакции базы данных, могут еще не быть отражены в базе данных. Кроме того, любые модели или записи базы данных, созданные в рамках транзакции, могут не существовать в базе данных. Если ваше событие зависит от этих моделей, могут возникнуть непредвиденные ошибки при обработке задания, транслирующего событие.

Если для параметра `after_commit` конфигурации вашего соединения с очередью установлено значение `false`, то вы все равно можете указать, что конкретное транслируемое событие должно быть отправлено после того, как все открытые транзакции базы данных были зафиксированы, реализовав интерфейс `ShouldDispatchAfterCommit` в классе события:

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Contracts\Events\ShouldDispatchAfterCommit;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast, ShouldDispatchAfterCommit
{
    use SerializesModels;
}
```

Чтобы узнать больше о том, как обойти эти проблемы, просмотрите документацию, касающуюся [заданий в очереди и транзакций базы данных](#).

## # Авторизация каналов

Частные каналы требуют, чтобы текущий аутентифицированный пользователь был авторизован и действительно мог прослушивать канал. Это достигается путем отправки HTTP-запроса вашему приложению Laravel с именем канала, что

позволит вашему приложению определить, может ли пользователь прослушивать этот канал. При использовании [Laravel Echo](#) HTTP-запрос на авторизацию подписок на частные каналы будет выполнен автоматически.

Когда вещание включено, Laravel автоматически регистрирует маршрут `/broadcasting/auth` для обработки запросов на авторизацию. Маршрут `/broadcasting/auth` автоматически помещается в группу посредников [web](#).

## Определение авторизации канала

Затем нам нужно определить логику, которая фактически будет определять, может ли текущий аутентифицированный пользователь прослушивать указанный канал. Это делается в файле `routes/channels.php`, созданном командой Artisan `install:broadcasting`. В этом файле вы можете использовать метод `Broadcast::channel` для регистрации замыканий авторизации канала:

```
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

Метод `channel` принимает два аргумента: имя канала и замыкание, которое возвращает `true` или `false`, указывая тем самым, имеет ли пользователь право прослушивать канал.

Все замыкания авторизации получают текущего аутентифицированного пользователя в качестве своего первого аргумента и любые дополнительные параметры в качестве своих последующих аргументов. В этом примере мы используем заполнитель `{orderId}`, чтобы указать, что часть «ID» имени канала является параметром.

Вы можете просмотреть список замыканий авторизации вещания вашего приложения, используя команду Artisan `channel:list`:

```
php artisan channel:list
```

## Привязка модели к авторизации

Как и HTTP-маршруты, для маршрутов каналов также могут использоваться неявные и явные [привязки модели к маршруту](#). Например, вместо получения строкового или числового идентификатора заказа вы можете запросить фактический экземпляр модели `Order`:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{order}', function (User $user, Order $order) {
    return $user->id === $order->user_id;
});
```

В отличие от привязки модели к HTTP-маршруту, привязка модели канала не поддерживает [ограничение неявной привязки модели](#). Однако это редко представляет собой проблему, потому что большинство каналов можно ограничить на основе уникального первичного ключа одной модели.

## Предварительная аутентификация авторизации канала

Частные каналы и каналы присутствия аутентифицируют текущего пользователя через стандартного охранника аутентификации вашего приложения. Если пользователь не аутентифицирован, то авторизация канала автоматически отклоняется, и обратный вызов авторизации никогда не выполняется. Однако вы можете назначить несколько своих охранников, которые должны при необходимости аутентифицировать входящий запрос:

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]));
```

## Определение класса канала

Если ваше приложение использует много разных каналов, то ваш файл `routes/channels.php` может стать громоздким. Таким образом, вместо использования замыканий для авторизации каналов вы можете использовать классы

каналов. Чтобы сгенерировать новый канал, используйте команду `make:channel Artisan`. Эта команда поместит новый класс канала в каталог `app/Broadcasting` вашего приложения:

```
php artisan make:channel OrderChannel
```

Затем зарегистрируйте свой канал в файле `routes/channels.php`:

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

Наконец, вы можете поместить логику авторизации для своего канала в метод `join` класса канала. Этот метод будет содержать ту же логику, которую вы обычно использовали бы в замыкании при авторизации вашего канала. Вы также можете воспользоваться преимуществами привязки модели канала:

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * Создать новый экземпляр канала.
     */
    public function __construct() {}

    /**
     * Подтвердить доступ пользователя к каналу.
     */
    public function join(User $user, Order $order): array|bool
    {
        return $user->id === $order->user_id;
    }
}
```

Как и многие другие классы в Laravel, классы каналов будут автоматически разрешены [контейнером служб](#). Таким образом, вы можете указать любые зависимости, необходимые для вашего канала, в его конструкторе.

## # Трансляция событий

После того как вы определили событие и отметили его интерфейсом `ShouldBroadcast`, вам нужно только запустить событие, используя метод отправки события. Диспетчер событий заметит, что событие помечено интерфейсом `ShouldBroadcast`, и поставит событие в очередь для дальнейшей трансляции:

```
use App\Events\OrderShipmentStatusUpdated;  
  
OrderShipmentStatusUpdated::dispatch($order);
```

## Трансляция событий только остальным пользователям

При создании приложения, использующего трансляцию событий, иногда может потребоваться трансляция события всем подписчикам канала, кроме текущего пользователя. Вы можете сделать это с помощью помощника `broadcast` и метода `toOthers`:

```
use App\Events\OrderShipmentStatusUpdated;  
  
broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

Чтобы лучше понять необходимость использования метода `toOthers`, давайте представим приложение со списком задач, в котором пользователь может создать новую задачу, введя имя задачи. Чтобы создать задачу, ваше приложение может сделать запрос к URL-адресу `/task`, который транслирует создание задачи и возвращает JSON-представление новой задачи. Когда ваше JavaScript-приложение

получает ответ от конечной точки, оно может напрямую вставить новую задачу в свой список задач следующим образом:

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
 });
```

Однако помните, что мы также транслируем создание задачи. Если ваше JavaScript-приложение также прослушивает это событие, чтобы добавить задачи в список задач, у вас будут дублирующиеся задачи в вашем списке: одна из конечной точки и одна из трансляции. Вы можете решить эту проблему, используя метод `toOthers`, чтобы указать вещателю не транслировать событие текущему пользователю.

Ваше событие должно использовать трейт  
`Illuminate\Broadcasting\InteractsWithSockets` для  
вызыва метода `toOthers`.

## Конфигурирование при использовании метода `toOthers`

Когда вы инициализируете экземпляр Laravel Echo, соединению назначается идентификатор сокета. Если вы используете глобальный экземпляр `Axios` для выполнения HTTP-запросов из вашего JavaScript-приложения, то идентификатор сокета будет автоматически прикрепляться к каждому исходящему запросу в заголовке `X-Socket-ID`. Затем, когда вы вызываете метод `toOthers`, Laravel извлечет идентификатор сокета из заголовка и проинструктирует вещателя не транслировать никакие соединения с этим идентификатором сокета.

Если вы не используете глобальный экземпляр `Axios`, то вам необходимо вручную сконфигурировать JavaScript-приложение для отправки заголовка `X-Socket-ID` со всеми исходящими запросами. Вы можете получить идентификатор сокета, используя метод `Echo.socketId`:

```
var socketId = Echo.socketId();
```

# Настройка подключения

Если ваше приложение взаимодействует с несколькими широковещательными соединениями, и вы хотите транслировать событие с использованием вещателя, отличного от используемого по умолчанию, вы можете указать, на какое соединение отправлять событие, используя метод `via`:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->via('pusher');
```

В качестве альтернативы вы можете указать широковещательное соединение события, вызвав метод `broadcastVia` в конструкторе события. Однако перед этим вы должны убедиться, что класс событий использует трейт `InteractsWithBroadcasting`:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithBroadcasting;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    use InteractsWithBroadcasting;

    /**
     * Create a new event instance.
     */
    public function __construct()
    {
        $this->broadcastVia('pusher');
    }
}
```

# Анонимные события

Иногда вам может потребоваться транслировать простое событие во внешний интерфейс вашего приложения без создания специального класса событий. Чтобы обеспечить это, фасад `Broadcast` позволяет транслировать «анонимные события»:

```
Broadcast::on('orders.' . $order->id) -> send();
```

В приведенном выше примере будет транслироваться следующее событие:

```
{
    "event": "AnonymousEvent",
    "data": "[]",
    "channel": "orders.1"
}
```

Используя методы `as` и `with`, вы можете настроить имя и данные события:

```
Broadcast::on('orders.' . $order->id)
    -> as('OrderPlaced')
    -> with($order)
    -> send();
```

В приведенном выше примере будет транслироваться событие, подобное следующему:

```
{
    "event": "OrderPlaced",
    "data": "{ id: 1, total: 100 }",
    "channel": "orders.1"
}
```

Если вы хотите транслировать анонимное событие на частном канале или канале присутствия, вы можете использовать методы `private` и `presence`:

```
Broadcast::private('orders.' . $order->id) -> send();
Broadcast::presence('channels.' . $channel->id) -> send();
```

При рассылке анонимного события с помощью метода `send` оно отправляется в очередь вашего приложения для обработки. Однако, если вы хотите немедленно транслировать событие, вы можете использовать метод `sendNow`:

```
Broadcast::on('orders.'.$order->id)->sendNow();
```

Чтобы транслировать событие всем подписчикам канала, кроме текущего аутентифицированного пользователя, вы можете вызвать метод `toOthers`:

```
Broadcast::on('orders.'.$order->id)
    ->toOthers()
    ->send();
```

## # Прием трансляций

### Прослушивание событий

После того как вы установили и создали экземпляр Laravel Echo, вы готовы к прослушиванию событий, которые транслируются из вашего приложения Laravel. Сначала используйте метод `channel` для получения экземпляра канала, затем вызовите метод `listen` для прослушивания конкретного события:

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
   });
```

Если вы хотите прослушивать события на частном канале, то используйте вместо этого метод `private`. Вы можете продолжить цепочку вызовов метода `listen` для прослушивания нескольких событий на одном канале:

```
Echo.private(`orders.${this.order.id}`)
    .listen(/* ... */)
    .listen(/* ... */)
    .listen(/* ... */);
```

## Остановка прослушивания событий

Если вы хотите прекратить прослушивание данного события не [покидая канал](#), вы можете использовать метод `stopListening`:

```
Echo.private(`orders.${this.order.id}`)
  .stopListening('OrderShipmentStatusUpdated')
```

## Покидание канала

Чтобы покинуть канал, вы можете вызвать метод `leaveChannel` вашего экземпляра Echo:

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

Если вы хотите покинуть канал, а также связанные с ним частные каналы и каналы присутствия, вы можете вызвать метод `leave`:

```
Echo.leave(`orders.${this.order.id}`);
```

## Пространства имён

Вы могли заметить в приведенных выше примерах, что мы не указали полное пространство имен `App\Events` для классов событий. Это связано с тем, что Echo автоматически предполагает, что события находятся в пространстве имен `App\Events`. Однако вы можете изменить корневое пространство имен при создании экземпляра Echo, передав параметр конфигурации `namespace`:

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  // ...
  namespace: 'App.Other.Namespace'
});
```

В качестве альтернативы вы можете добавить к классам событий префикс `.` при подписке на них с помощью Echo. Это позволит вам всегда указывать полное имя класса:

```
Echo.channel('orders')
  .listen('.Namespace\\Event\\Class', (e) => {
    // ...
  });
});
```

## # Каналы присутствия

Каналы присутствия основаны на безопасности частных каналов, но в то же время предоставляя дополнительную функцию осведомленности о том, кто подписан на канал. Это упрощает создание мощных функций приложения для совместной работы, таких как уведомление пользователей, когда другой пользователь просматривает ту же страницу, или перечисление пользователей комнаты чата.

## Авторизация каналов присутствия

Все каналы присутствия также являются частными; следовательно, пользователи должны быть [авторизованы для доступа к ним](#). Однако при определении замыканий авторизации для каналов присутствия вы не должны возвращать `true`, если пользователь авторизован для присоединения к каналу. Вместо этого вы должны вернуть массив данных о пользователе.

Данные, возвращаемые замыканием авторизации, будут доступны для слушателей событий канала присутствия в вашем JavaScript-приложении. Если пользователь не авторизован для присоединения к каналу присутствия, то вы должны вернуть `false` или `null`:

```
use App\Models\User;

Broadcast::channel('chat.{roomId}', function (User $user, int $roomId) {
  if ($user->canJoinRoom($roomId)) {
    return ['id' => $user->id, 'name' => $user->name];
  }
});
```

## Присоединение к каналам присутствия

Чтобы присоединиться к каналу присутствия, вы можете использовать метод `join` Echo. Метод `join` вернет реализацию `PresenceChannel`, которая, наряду с методом `listen`, позволяет вам подписаться на события `here`, `joining` и `leave`.

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        // ...
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    })
    .error((error) => {
        console.error(error);
});
});
```

Замыкание `here` будет выполнено сразу после успешного присоединения к каналу и получит массив, содержащий информацию о пользователе для всех других пользователей, которые в настоящее время подписаны на канал. Метод `joining` будет выполняться, когда новый пользователь присоединяется к каналу, а метод `leaving` будет выполняться при покидании пользователем канала. Метод `error` будет выполнен, когда при аутентификации возвращается код HTTP-статуса отличный от 200 или если есть проблема с парсингом возвращаемого JSON.

## Трансляция на каналы присутствия

Каналы присутствия могут получать события так же, как публичные или частные каналы. Используя пример чата, мы можем захотеть транслировать события `NewMessage` на канал присутствия комнаты. Для этого мы вернем экземпляр `PresenceChannel` из метода `broadcastOn` события:

```
/**
 * Получить каналы трансляции события.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

Как и в случае с другими событиями, вы можете использовать помощник `broadcast` и метод `toOthers`, чтобы исключить текущего пользователя из приема трансляции:

```
broadcast(new NewMessage($message));  
  
broadcast(new NewMessage($message))->toOthers();
```

Как и для других типов событий, вы можете прослушивать события, отправленные в каналы присутствия, используя метод `listen` Echo:

```
Echo.join(`chat.${roomId}`)  
.here(/* ... */)  
.joining(/* ... */)  
.leaving(/* ... */)  
.listen('NewMessage', (e) => {  
    // ...  
});
```

## # Трансляция моделей

Прежде чем читать следующую документацию о трансляции моделей, мы рекомендуем вам ознакомиться с общими концепциями модельных широковещательных служб Laravel, а также с тем, как вручную создавать и прослушивать широковещательные события.

Обычно транслируются события, когда [модели Eloquent](#) создаются, обновляются или удаляются. Конечно, это легко можно сделать вручную, [определив пользовательские события для изменений состояния модели Eloquent](#) и пометив эти события с помощью интерфейса `ShouldBroadcast`.

Однако, если вы не используете эти события для каких-либо других целей в своем приложении, может оказаться обременительным создание классов событий с единственной целью их широковещательной передачи. Чтобы исправить это, Laravel позволяет вам указать, что модель Eloquent должна автоматически транслировать изменения своего состояния.

Для начала ваша модель Eloquent должна использовать трейт `Illuminate\Database\Eloquent\BroadcastsEvents`. Кроме того, модель должна определять метод `broadcastOn`, возвращающий массив каналов, по которым должны транслироваться события модели:

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use BroadcastsEvents, HasFactory;

    /**
     * Получите пользователя, которому принадлежит сообщение.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }

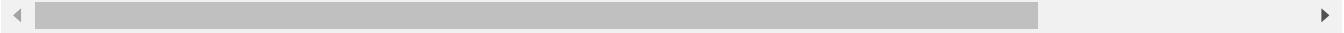
    /**
     * Получите каналы, по которым должны транслироваться события модели.
     *
     * @return array<int, \Illuminate\Broadcasting\Channel|\Illuminate\Database\Eloquent\BroadcastsEvents>
     */
    public function broadcastOn(string $event): array
    {
        return [$this, $this->user];
    }
}
```

После того как ваша модель включает этот трейт и определяет свои каналы вещания, она начнет автоматически транслировать события при создании, обновлении, удалении, уничтожении или восстановлении экземпляра модели.

Кроме того, вы могли заметить, что метод `broadcastOn` получает строковый аргумент `$event`. Этот аргумент содержит тип события, которое произошло в модели, и будет

иметь значение `created`, `updated`, `deleted`, `trashed` ИЛИ `restored`. Проверяя значение этой переменной, вы можете определить, на какие каналы (если есть) модель должна транслировать конкретное событие:

```
/*
 * Получите каналы, по которым должны транслироваться события модели.
 *
 * @return array<string, array<int, \Illuminate\Broadcasting\Channel|\Illuminate\Da
 */
public function broadcastOn(string $event): array
{
    return match ($event) {
        'deleted' => [],
        default => [$this, $this->user],
    };
}
```

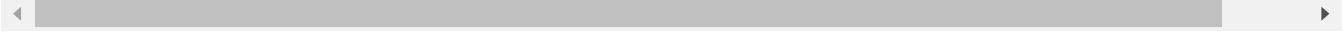


## Настройка создания события трансляции модели

Иногда вы можете захотеть настроить то, как Laravel создает базовое событие трансляции модели. Вы можете добиться этого, определив метод `newBroadcastableEvent` в вашей модели Eloquent. Этот метод должен возвращать экземпляр `\Illuminate\Database\Eloquent\BroadcastableModelEventOccurred`:

```
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred;

/**
 * Создайте новое транслируемое событие для модели.
 */
protected function newBroadcastableEvent(string $event): BroadcastableModelEventOccu
{
    return (new BroadcastableModelEventOccurred(
        $this, $event
    ))->dontBroadcastToCurrentUser();
}
```



## Соглашение о трансляции моделей

### Соглашения о каналах

Как вы могли заметить, метод `broadcastOn` в приведенном выше примере модели не возвращал экземпляры `Channel`. Вместо этого модели Eloquent возвращались напрямую. Если экземпляр модели Eloquent возвращается методом `broadcastOn` вашей модели (или содержится в массиве, возвращаемом методом), Laravel автоматически создаст экземпляр частного канала для модели, используя имя класса модели и идентификатор первичного ключа в качестве названия канала.

Итак, модель `App\Models\User` с `id` равным `1` будет преобразована в экземпляр `Illuminate\Broadcasting\PrivateChannel` с именем `App.Models.User.1`. Конечно, в дополнение к возврату экземпляров модели Eloquent из метода `broadcastOn` вашей модели, вы можете возвращать полные экземпляры `Channel` чтобы иметь полный контроль над именами каналов модели:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Получите каналы, по которым должны транслироваться события модели.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(string $event): array
{
    return [
        new PrivateChannel('user.'.$this->id)
    ];
}
```

Если вы планируете явно возвращать экземпляр канала из метода `broadcastOn` вашей модели, вы можете передать экземпляр модели Eloquent в конструктор канала. При этом Laravel будет использовать описанные выше соглашения о каналах модели, чтобы преобразовать модель Eloquent в строку имени канала:

```
return [new Channel($this->user)];
```

Если вам нужно определить имя канала модели, вы можете вызвать метод `broadcastChannel` для любого экземпляра модели. Например, этот метод возвращает строку `App.Models.User.1` для модели `App\Models\User` с `id` равным `1`:

```
$user->broadcastChannel()
```

## Соглашение о событиях

Поскольку события трансляции моделей не связаны с «фактическим» событием в каталоге `App\Events` вашего приложения, им присваиваются имя и полезная нагрузка на основе соглашений. Соглашение Laravel состоит в том, чтобы транслировать событие, используя имя класса модели (не включая пространство имен) и имя события модели, которое инициировало трансляцию.

Так, например, обновление модели `App\Models\Post` будет транслировать событие в ваше клиентское приложение как `PostUpdated` со следующей полезной нагрузкой:

```
{  
    "model": {  
        "id": 1,  
        "title": "My first post"  
        ...  
    },  
    ...  
    "socket": "someSocketId",  
}
```

Удаление модели `App\Models\User` приведет к трансляции события с именем `UserDeleted`.

При желании вы можете определить собственное имя трансляции и полезную нагрузку, добавив к вашей модели методы `broadcastAs` и `broadcastWith`. Эти методы получают имя происходящего события / операции модели, что позволяют вам настроить имя события и полезную нагрузку для каждой операции модели. Если `null` возвращается из метода `broadcastAs`, Laravel будет использовать соглашения об именах широковещательных событий модели, обсужденные выше:

```
/**  
 * The model event's broadcast name.  
 */  
public function broadcastAs(string $event): string|null  
{  
    return match ($event) {  
        'created' => 'post.created',  
        default => null,  
    };  
}
```

```
/**  
 * Get the data to broadcast for the model.  
 *  
 * @return array<string, mixed>  
 */  
public function broadcastWith(string $event): array  
{  
    return match ($event) {  
        'created' => ['title' => $this->title],  
        default => ['model' => $this],  
    };  
}
```

## Прослушивание трансляций моделей

После того как вы добавили в модель трейт `BroadcastsEvents` и определили метод `broadcastOn` модели, вы готовы начать прослушивание транслируемых событий модели в своем клиентском приложении. Перед тем как начать, вы можете ознакомиться с полной документацией по [прослушиванию событий](#).

Сначала используйте метод `private` для получения экземпляра канала, затем вызовите метод `listen` для прослушивания указанного события. Как правило, имя канала, присвоенное методу `private` должно соответствовать [соглашению о трансляции моделей](#).

Как только вы получили экземпляр канала, вы можете использовать метод `listen` для прослушивания определенного события. Поскольку события трансляции моделей не связаны с «фактическим» событием в каталоге вашего приложения `App\\Events` directory, [имя события](#) должно иметь префикс `.,` чтобы указать, что оно не принадлежит определенному пространству имен. Каждое событие трансляции модели имеет свойство `model`, которое содержит все транслируемые свойства модели:

```
Echo.private(`App.Models.User.${this.user.id}`)  
    .listen('.PostUpdated', (e) => {  
        console.log(e.model);  
    });
```

## # Клиентские события

При использовании [Pusher Channels](#) вы должны включить опцию «Client Events» в разделе «App Settings» вашей [панели управления приложения](#) для отправки клиентских событий.

По желанию можно транслировать событие другим подключенными клиентам, вообще не затрагивая ваше приложение Laravel. Это может быть особенно полезно для таких вещей, как «ввод» уведомлений, когда вы хотите предупредить пользователей вашего приложения о том, что другой пользователь печатает сообщение.

Чтобы транслировать клиентские события, вы можете использовать метод [whisper](#) Echo:

```
Echo.private(`chat.${roomId}`)
  .whisper('typing', {
    name: this.user.name
 });
```

Чтобы прослушивать клиентские события, вы можете использовать метод [listenForWhisper](#):

```
Echo.private(`chat.${roomId}`)
  .listenForWhisper('typing', (e) => {
    console.log(e.name);
 });
```

## # Уведомления

Если связать трансляцию событий с [уведомлениями](#), то ваше JavaScript-приложение может получать новые уведомления по мере их появления без необходимости в обновлении страницы. Перед началом работы обязательно прочтите документацию по использованию [канала транслируемых уведомлений](#).

После того как вы настроили уведомление для использования трансляции канала, вы можете прослушивать транслируемые события, используя метод [notification](#)

Echo. Помните, что имя канала должно соответствовать имени класса объекта, получающего уведомления:

```
Echo.private(`App.Models.User.${userId}`)
.notification((notification) => {
    console.log(notification.type);
});
```

В этом примере все уведомления, отправленные экземплярам `App\Models\User` через канал `broadcast`, будут получены в замыкании. Авторизация канала `App.Models.User.{id}` включена в файл `routes/channels.php` вашего приложения.

# Кэширование

## # Введение

## # Конфигурирование

# Предварительная подготовка драйверов

## # Управление кешем приложения

# Получение экземпляра кеша

# Получение элементов из кеша

# Сохранение элементов в кеше

# Удаление элементов из кеша

# Глобальный помощник кеша

## # Атомарные блокировки

# Управление блокировками

# Управление блокировками между процессами

## # Добавление собственных драйверов кеша

# Написание драйвера кеша

# Регистрация драйвера кеша

## # События

## # Введение

Некоторые задачи по извлечению или обработке данных, выполняемые вашим приложением, могут потребовать больших ресурсов ЦП или занять несколько секунд. В этом случае извлеченные данные обычно кешируют на некоторое время, чтобы их можно было быстро извлечь при последующих запросах тех же данных. Кешированные данные обычно хранятся в хранилище с быстрым доступом данных, например, [Memcached](#) или [Redis](#).

К счастью, Laravel предлагает выразительный унифицированный API для различных серверов кеширования, позволяя вам воспользоваться их невероятно быстрым извлечением данных и ускорить работу вашего веб-приложения.

# # Конфигурирование

Файл конфигурации кеша вашего приложения находится в `config/cache.php`. В этом файле вы можете указать, какой хранилище кеша вы хотите использовать по умолчанию для всего приложения. Laravel из коробки поддерживает популярные механизмы кеширования, такие как [Memcached](#), [Redis](#), [DynamoDB](#) и реляционные базы данных. Кроме того, доступен драйвер кеширования на основе файлов, в то время как драйверы `array` и `null` предоставляют удобные механизмы кеширования для ваших автоматических тестов.

Файл конфигурации кэша также содержит множество других параметров, которые вы можете просмотреть. По умолчанию Laravel настроен на использование драйвера кэша `database`, который сохраняет сериализованные кэшированные объекты в базе данных вашего приложения.

## Предварительная подготовка драйверов

### Предварительная подготовка драйвера на основе базы данных

При использовании драйвера кэша `database` вам понадобится таблица базы данных, содержащая данные кэша. Обычно это включено в стандартный файл Laravel `0001_01_01_000001_create_cache_table.php` [миграции базы данных](#); однако, если ваше приложение не содержит этой миграции, вы можете использовать Artisan-команду `make:cache-table` для ее создания:

```
php artisan make:cache-table
```

```
php artisan migrate
```

### Предварительная подготовка драйвера на основе Memcached

Для использования драйвера Memcached требуется установить [пакет Memcached PECL](#). Вы можете перечислить все ваши серверы Memcached в файле конфигурации `config/cache.php`. Этот файл уже содержит запись `memcached.servers` для начала:

```
'memcached' => [
    // ...

    'servers' => [
        [
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),
            'port' => env('MEMCACHED_PORT', 11211),
            'weight' => 100,
        ],
    ],
],
],
```

При необходимости вы можете задать параметр `host` сокета UNIX. Если вы это сделаете, то параметр `port` должен быть задан как `0`:

```
'memcached' => [
    [
        // ...
    ]

    'servers' => [
        [
            'host' => '/var/run/memcached/memcached.sock',
            'port' => 0,
            'weight' => 100
        ],
    ],
],
],
```

## Предварительная подготовка драйвера на основе Redis

Перед использованием драйвера кеша Redis, вам нужно будет либо установить расширение PHP `PhpRedis` через PECL, либо установить пакет `predis/predis` (~ 2.0) через Composer. [Laravel Sail](#) уже включает это расширение. Кроме того, на официальных платформах развертывания Laravel, таких как [Laravel Forge](#) и [Laravel Vapor](#), расширение `PhpRedis` установлено по умолчанию.

Для получения дополнительной информации о настройке Redis обратитесь к его [странице документации Laravel](#).

## Предварительная подготовка драйвера на основе DynamoDB

Перед использованием драйвера кэша [DynamoDB](#) необходимо создать таблицу

DynamoDB для хранения всех кэшированных данных. Обычно это `cache`.

Название таблицы должно совпадать с `stores.dynamodb.table` в конфигурационном файле `cache`. Имя таблицы также можно задать с помощью переменной среды `DYNAMODB_CACHE_TABLE`.

Эта таблица также должна иметь строковый ключ раздела с именем, соответствующим значению элемента конфигурации `stores.dynamodb.attributes.key` в конфигурационном файле `cache`. По умолчанию это `key`.

Обычно DynamoDB не удаляет элементы с истекшим сроком действия из таблицы заранее. Поэтому вам следует [включить Time to Live \(TTL\)](#) на таблице. При настройке параметров TTL таблицы вам следует установить имя атрибута TTL на `expires_at`.

Затем установите AWS SDK, чтобы ваше приложение Laravel могло взаимодействовать с DynamoDB:

```
composer require aws/aws-sdk-php
```

Кроме того, вам следует убедиться, что указаны значения для параметров конфигурации хранилища кэша DynamoDB. Обычно эти параметры, такие как `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY`, должны быть определены в файле конфигурации `.env` вашего приложения:

```
'dynamodb' => [
    'driver' => 'dynamodb',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => env('DYNAMODB_CACHE_TABLE', 'cache'),
    'endpoint' => env('DYNAMODB_ENDPOINT'),
],
```

## # Управление кешем приложения

### Получение экземпляра кеша

Чтобы получить экземпляр хранилища кеша, вы можете использовать фасад `Cache`, который мы будем использовать в этой документации. Фасад `Cache` обеспечивает удобный и краткий доступ к базовым реализациям контрактов кеширования Laravel:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Показать список всех пользователей приложения.
     */
    public function index(): array
    {
        $value = Cache::get('key');

        return [
            // ...
        ];
    }
}
```

## Доступ к различным кеш-хранилищам

Используя фасад `Cache`, вы можете получить доступ к различным хранилищам кеша с помощью метода `store`. Ключ, переданный методу `store`, должен соответствовать одному из хранилищ, перечисленных в массиве `stores` вашего конфигурационного файла `config/cache.php`:

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

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

## Получение элементов из кеша

Метод `get` фасада `Cache` используется для извлечения элементов из кеша. Если элемент не существует в кеше, будет возвращено значение `null`. Если хотите, то вы можете передать второй аргумент методу `get`, указав значение по умолчанию, которое вы хотите вернуть, если элемент отсутствует:

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

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

Вы даже можете передать замыкание в качестве значения по умолчанию. Результат замыкания будет возвращен, если указанный элемент не существует в кеше. Передача замыкания позволяет отложить получение значений по умолчанию из базы данных или другой внешней службы:

```
$value = Cache::get('key', function () {
    return DB::table(/* ... */)->get();
});
```

## Проверка наличия элемента

Метод `has` используется для определения того, существует ли элемент в кеше. Этот метод также вернет `false`, если элемент существует, но его значение равно `null`:

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

## Увеличение и уменьшение отдельных значений в кеше

Методы `increment` и `decrement` могут использоваться для изменения значений целочисленных элементов в кеше. Оба метода принимают необязательный второй аргумент, указывающий величину увеличения или уменьшения значения элемента:

```
// Initialize the value if it does not exist...
Cache::add('key', 0, now()->addHours(4));

// Increment or decrement the value...
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

# Выполнение замыкания с последующим сохранением и получением результата

Также вы можете не только получить элемент из кеша, но и сохранить значение по умолчанию, если запрошенный элемент не существует. Например, вы можете получить всех пользователей из кеша или, если они не существуют, получить их из базы данных и добавить их в кеш. Вы можете сделать это с помощью метода `Cache::remember`:

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

Если элемент не существует в кеше, то замыкание, переданное методу `remember`, будет выполнено, и его результат будет помещен в кеш.

Вы можете использовать метод `rememberForever`, чтобы получить элемент из кеша или сохранить его навсегда, если он не существует:

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

## Устарело при повторной проверке

При использовании метода `Cache::remember` у некоторых пользователей может наблюдаться медленное время отклика, если срок действия кэшированного значения истек. Для определенных типов данных может быть полезно разрешить обслуживание частично устаревших данных во время пересчета кэшированного значения в фоновом режиме, чтобы у некоторых пользователей не возникало медленного ответа во время расчета кэшированных значений. Это часто называют шаблоном «устаревшего при повторной проверке» ("stale-while-revalidate"), и метод `Cache::flexible` обеспечивает реализацию этого шаблона.

Гибкий метод принимает массив, который определяет, как долго кэшированное значение считается «свежим», а когда оно становится «устаревшим». Первое значение в массиве представляет количество секунд, в течение которых кеш считается свежим, а второе значение определяет, как долго он может использоваться как устаревшие данные, прежде чем потребуется пересчет.

Если запрос сделан в свежем периоде (до первого значения), кэш возвращается сразу без пересчета. Если запрос сделан в течение периода устаревания (между двумя значениями), устаревшее значение передается пользователю, и [отложенная функция](#) регистрируется в обновить кэшированное значение после отправки ответа пользователю. Если запрос сделан после второго значения, кеш считается просроченным, и значение немедленно пересчитывается, что может привести к более медленному ответу пользователя:

```
$value = Cache::flexible('users', [5, 10], function () {
    return DB::table('users')->get();
});
```

## Получение данных с последующим удалением элемента

Если вам нужно получить элемент из кеша, а затем удалить этот элемент, вы можете использовать метод `pull`. Как и в методе `get`, если элемент не существует в кеше, то будет возвращен `null`:

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

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

## Сохранение элементов в кеше

Вы можете использовать метод `put` фасада `Cache` для сохранения элементов в кеше:

```
Cache::put('key', 'value', $seconds = 10);
```

Если время хранения не передается методу `put`, то элемент будет храниться бесконечно:

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

Вместо того чтобы передавать количество секунд как целое число, вы также можете передать экземпляр `DateTime`, представляющий желаемое время хранения кэшированного элемента:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

## Сохранение значений при условии их отсутствия

Метод `add` добавит элемент в кеш, только если он еще не существует в хранилище кеша. Метод вернет `true`, если элемент был действительно добавлен в кеш. В противном случае метод вернет `false`. Метод `add` – это [атомарная операция](#):

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

## Сохранение элементов на постоянной основе

Метод `forever` используется для постоянного хранения элемента в кеше. Поскольку срок действия этих элементов не истекает, то их необходимо вручную удалить из кеша с помощью метода `forget`:

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

Если вы используете драйвер `memcached`, то элементы, которые хранятся «на постоянной основе», могут быть удалены, когда кеш достигнет предельного размера.

## Удаление элементов из кеша

Вы можете удалить элементы из кеша с помощью метода `forget`:

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

Вы также можете удалить элементы, указав нулевое или отрицательное количество секунд срока хранения:

```
Cache::put('key', 'value', 0);
```

```
Cache::put('key', 'value', -5);
```

Вы можете очистить весь кеш, используя метод `flush`:

```
Cache::flush();
```

Очистка кеша не учитывает ваш настроенный «префикс» кеша и удаляет все записи из кеша. Внимательно учитывайте это при очистке кеша, который используется другими приложениями.

## Глобальный помощник кеша

Помимо использования фасада `Cache`, вы также можете использовать глобальную функцию `cache` для извлечения и хранения данных через кеш. Когда функция `cache` вызывается с одним строковым аргументом, она возвращает значение переданного ключа:

```
$value = cache('key');
```

Если вы передадите массив пар ключ / значение и срок хранения в функцию, то она будет хранить значения в кеше в течение указанного времени:

```
cache(['key' => 'value'], $seconds);  
cache(['key' => 'value'], now()->addMinutes(10));
```

Когда функция `cache` вызывается без каких-либо аргументов, то она возвращает экземпляр реализации `Illuminate\Contracts\Cache\Factory`, позволяя вам вызывать другие методы кеширования:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

При тестировании вызова глобальной функции `cache` вы можете использовать метод `Cache::shouldReceive` так же, как если бы вы [тестировали фасад](#).

## # Атомарные блокировки

Чтобы использовать этот функционал, ваше приложение должно использовать драйвер кеша `memcached`, `redis`, `dynamodb`, `database`, `file`, или `array` в качестве драйвера кеша по умолчанию для вашего приложения. Кроме того, все серверы должны взаимодействовать с одним и тем же центральным сервером кеширования.

## Управление блокировками

Атомарные блокировки позволяют управлять распределенными блокировками, не беспокоясь об условиях приоритетности. Например, [Laravel Forge](#) использует атомарные блокировки, чтобы гарантировать, что на сервере одновременно выполняется только одна удаленная задача. Вы можете создавать и управлять блокировками, используя метод `Cache::lock`:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Блокировка получена на 10 секунд ...
```

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

Метод `get` также принимает замыкание. После выполнения замыкания Laravel автоматически снимет блокировку:

```
Cache::lock('foo', 10)->get(function () {  
    // Блокировка установлена на 10 секунд и автоматически снимается ...  
});
```

Если блокировка недоступна в тот момент, когда вы ее запрашиваете, вы можете указать Laravel подождать определенное количество секунд. Если блокировка не может быть получена в течение указанного срока, то будет выброшено исключение `Illuminate\Contracts\Cache\LockTimeoutException`:

```
use Illuminate\Contracts\Cache\LockTimeoutException;  
  
$lock = Cache::lock('foo', 10);  
  
try {  
    $lock->block(5);  
  
    // Блокировка получена после ожидания максимум 5 секунд ...  
} catch (LockTimeoutException $e) {  
    // Невозможно получить блокировку ...  
} finally {  
    $lock->release();  
}
```

Приведенный выше пример можно упростить, передав замыкание методу `block`. Когда замыкание передается этому методу, Laravel будет пытаться получить блокировку на указанное количество секунд и автоматически снимет блокировку, как только замыкание будет выполнено:

```
Cache::lock('foo', 10)->block(5, function () {  
    // Блокировка получена после ожидания максимум 5 секунд ...  
});
```

## Управление блокировками между процессами

Иногда может потребоваться установить блокировку в одном процессе и снять ее в другом процессе. Например, вы можете получить блокировку во время веб-запроса и захотите снять блокировку в конце задания в очереди, которое запускается этим запросом. В этом сценарии вы должны передать «токен инициатора» с областью действия блокировки в задании в очереди, чтобы задание могло повторно создать экземпляр блокировки с использованием данного токена.

В приведенном ниже примере мы отправим задание в очередь, если блокировка будет успешно получена. Кроме того, мы передадим токен инициатора блокировки заданию в очереди с помощью метода `owner` блокировки:

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

В рамках задания `ProcessPodcast` нашего приложения мы можем восстановить и снять блокировку с помощью токена инициатора:

```
Cache::restoreLock('processing', $this->owner)->release();
```

Если вы хотите принудительно снять блокировку без учета текущего инициатора, то вы можете использовать метод `forceRelease`:

```
Cache::lock('processing')->forceRelease();
```

## # Добавление собственных драйверов кеша

### Написание драйвера кеша

Чтобы создать собственный драйвер кеша, сначала нужно реализовать [контракт Illuminate\Contracts\Cache\Store](#). Итак, реализация кеша MongoDB может выглядеть примерно так:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}

}
```

Нам просто нужно реализовать каждый из этих методов, используя соединение MongoDB. Для примера того, как реализовать каждый из этих методов, взгляните на [Illuminate\Cache\MemcachedStore](#) в [исходном коде фреймворка Laravel](#). Как только наша реализация будет завершена, мы можем завершить регистрацию своего драйвера, вызвав метод `extend` фасада `Cache`:

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

Если вам интересно, где разместить свой собственный код драйвера кеша, то вы можете создать пространство имен `Extensions` в своем каталоге `app`. Однако имейте в виду, что Laravel не имеет жесткой структуры приложения, и вы можете организовать свое приложение в соответствии со своими предпочтениями.

## Регистрация драйвера кеша

Чтобы зарегистрировать свой драйвер кеша в Laravel, мы будем использовать метод `extend` фасада `Cache`. Поскольку другие поставщики служб могут попытаться прочитать кешированные значения в рамках своего метода `boot`, мы зарегистрируем свой драйвер в замыкании `booting`. Используя замыкание `booting`, мы можем гарантировать, что наш драйвер зарегистрирован непосредственно перед тем, как метод `boot` вызывается поставщиками служб нашего приложения, и после того, как метод `register` вызывается для всех поставщиков служб. Мы зарегистрируем наше замыкание `booting` в методе `register` класса `App\Providers\AppServiceProvider` нашего приложения:

```
<?php
```

```
namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        $this->app->booting(function () {
            Cache::extend('mongo', function (Application $app) {
                return Cache::repository(new MongoStore());
            });
        });
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        // ...
    }
}
```

Первым аргументом, передаваемым методу `extend`, является имя драйвера. Оно будет соответствовать вашему параметру `driver` в файле конфигурации

`config/cache.php`. Второй аргумент – это замыкание, которое должно возвращать экземпляр `Illuminate\Cache\Repository`. Замыкание будет передано экземпляру `$app`, который является экземпляром [контейнера служб](#).

После регистрации расширения обновите переменную среды `CACHE_STORE` или опцию `default` в файле конфигурации вашего приложения `config/cache.php`, указав имя вашего расширения.

## # События

Чтобы выполнить код при каждой операции с кэшем, вы можете прослушивать [события](#), запускаемые кэшем:

### Наименование события

`Illuminate\Cache\Events\CacheHit`

`Illuminate\Cache\Events\CacheMissed`

`Illuminate\Cache\Events\KeyForgotten`

`Illuminate\Cache\Events\KeyWritten`

Чтобы повысить производительность, вы можете отключить события кэширования, установив для параметра конфигурации `events` значение `false` для данного хранилища кэша в файле конфигурации `config/cache.php` вашего приложения:

```
'database' => [
    'driver' => 'database',
    // ...
    'events' => false,
],
```

# Коллекции

## # Введение

- # Создание коллекций
  - # Расширение коллекций
- ## # Доступные методы
- ## # Список методов
- ## # Сообщения высшего порядка
- ## # Отложенные коллекции
- # Введение в отложенные коллекции
  - # Создание отложенных коллекций
  - # Контракт Enumerable
  - # Методы отложенных коллекций

## # Введение

Класс `Illuminate\Support\Collection` обеспечивает гибкую и удобную обертку для работы с массивами данных. Например, посмотрите на следующий код. Здесь мы будем использовать хелпер `collect`, чтобы создать новый экземпляр коллекции из массива, запустим функцию `strtoupper` для каждого элемента, а затем удалим все пустые элементы:

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

Как видите, класс `Collection` позволяет объединять необходимые вам методы в цепочку для выполнения последовательного перебора и сокращения базового массива. В основном коллекции неизменяемы, то есть каждый метод коллекции возвращает совершенно новый экземпляр `Collection`.

# Создание коллекций

Как упоминалось выше, помощник `collect` возвращает новый экземпляр `Illuminate\Support\Collection` для переданного массива. Итак, создать коллекцию очень просто:

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

Результаты запросов `Eloquent` всегда возвращаются как экземпляры `Collection`.

## Расширение коллекций

Класс `Collection` являются «макропрограммируемым», что позволяет вам добавлять дополнительные методы к классу во время выполнения. Метод `macro` класса `Illuminate\Support\Collection` принимает функцию, которая будет выполнена при вызове вашего макроса. Эта функция может обращаться к другим методам коллекции через `$this`, как если бы это был реальный метод класса коллекции. Например, следующий код добавляет метод `toUpper` классу `Collection`:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function (string $value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Обычно макросы коллекций объявляются в методе `boot` [сервис-провайдера](#).

## Макросы с аргументами

При необходимости вы можете определить макросы, которые принимают дополнительные аргументы:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;

Collection::macro('toLocale', function (string $locale) {
    return $this->map(function (string $value) use ($locale) {
        return Lang::get($value, [], $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

## # Доступные методы

В большей части оставшейся документации по коллекциям мы обсудим каждый метод, доступный в классе `Collection`. Помните, что все эти методы можно объединить в цепочку для последовательного управления базовым массивом. Более того, почти каждый метод возвращает новый экземпляр `Collection`, позволяя вам при необходимости сохранить исходную копию коллекции:

[after](#)

[all\(\)](#)

[average\(\)](#)

[avg\(\)](#)

[before](#)

[chunk\(\)](#)

[chunkWhile\(\)](#)

[collapse\(\)](#)

[combine\(\)](#)

[collect\(\)](#)

[concat\(\)](#)

[contains\(\)](#)

containsOneItem()

containsStrict()

count()

countBy()

crossJoin()

dd()

diff()

diffAssoc()

diffAssocUsing()

diffKeys()

doesntContain

dot()

dump()

duplicates()

duplicatesStrict()

each()

eachSpread()

ensure()

every()

except()

filter()

first()

firstOrFail()

firstWhere()

flatMap()

flatten()

flip()

forget()

forPage()

get()

groupBy()

has()

hasAny()

implode()

intersect()  
intersectAssoc()  
intersectByKeys()  
isEmpty()  
isNotEmpty()  
join()  
keyBy()  
keys()  
last()  
lazy()  
macro()  
make()  
map()  
mapInto()  
mapSpread()  
mapToGroups()  
mapWithKeys()  
max()  
median()  
merge()  
mergeRecursive()  
min()  
mode()  
multiply  
nth()  
only()  
pad()  
partition()  
percentage()  
pipe()  
pipeInto()  
pipeThrough()  
pluck()  
pop()

prepend()  
pull()  
push()  
put()  
random()  
range  
reduce()  
reduceSpread  
reject()  
replace()  
replaceRecursive()  
reverse()  
search()  
select  
shift()  
shuffle()  
skip()  
skipUntil()  
skipWhile()  
slice()  
sliding  
sole  
some()  
sort()  
sortBy()  
sortByDesc()  
sortDesc()  
sortKeys()  
sortKeysDesc()  
sortKeysUsing()  
splice()  
split()  
splitIn()  
sum()

take()  
takeUntil()  
takeWhile()  
tap()  
times()  
toArray()  
toJson()  
transform()  
undot()  
union()  
unique()  
uniqueStrict()  
unless()  
unlessEmpty()  
unlessNotEmpty()  
unwrap()  
value()  
values()  
when()  
whenEmpty()  
whenNotEmpty()  
where()  
whereStrict()  
whereBetween()  
whereIn()  
whereInStrict()  
whereInstanceOf()  
whereNotBetween()  
whereNotIn()  
whereNotInStrict()  
whereNotNull()  
whereNull()  
wrap()

[zip\(\)](#)

## # Список методов

### after()

Метод `after` возвращает элемент после данного элемента. `null` возвращается, если данный элемент не найден или является последним элементом:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->after(3);  
  
// 4  
  
$collection->after(5);  
  
// null
```

Этот метод ищет данный элемент, используя «свободное» сравнение, то есть строка, содержащая целочисленное значение, будет считаться равной целому числу того же значения. Чтобы использовать «строгое» сравнение, вы можете предоставить методу аргумент `strict`:

```
collect([2, 4, 6, 8])->after('4', strict: true);  
  
// null
```

В качестве альтернативы вы можете предоставить собственное замыкание для поиска первого элемента, который проходит заданный тест на истинность:

```
collect([2, 4, 6, 8])->after(function (int $item, int $key) {  
    return $item > 5;  
});  
  
// 8
```

### all()

Метод `all` возвращает базовый массив, представленный коллекцией:

```
collect([1, 2, 3])->all();  
// [1, 2, 3]
```

## average()

Псевдоним для метода `avg`.

## avg()

Метод `avg` возвращает среднее значение переданного ключа:

```
$average = collect([  
    ['foo' => 10],  
    ['foo' => 10],  
    ['foo' => 20],  
    ['foo' => 40]  
])->avg('foo');  
  
// 20  
  
$average = collect([1, 1, 2, 4])->avg();  
  
// 2
```

## before()

Метод `before` является противоположностью метода `after`. Он возвращает элемент перед данным элементом. `null` возвращается, если данный элемент не найден или является первым элементом:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->before(3);  
  
// 2  
  
$collection->before(1);  
  
// null
```

```
collect([2, 4, 6, 8])->before('4', strict: true);

// null

collect([2, 4, 6, 8])->before(function (int $item, int $key) {
    return $item > 5;
});

// 4
```

## chunk()

Метод `chunk` разбивает коллекцию на несколько меньших коллекций указанного размера:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

// [[1, 2, 3, 4], [5, 6, 7]]
```

Этот метод особенно полезен в [шаблонах](#) при работе с сеткой, такой как [Bootstrap](#). Например, представьте, что у вас есть коллекция моделей [Eloquent](#), которые вы хотите отобразить в сетке:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

## chunkWhile()

Метод `chunkWhile` разбивает коллекцию на несколько меньших по размеру коллекций на основе результата переданного замыкания. Переменная `$chunk`, переданная в замыкание, может использоваться для проверки предыдущего элемента:

```
$collection = collect(str_split('AABBCCCD'));

$chunks = $collection->chunkWhile(function (string $value, int $key, Collection $chunk) {
    return $value === $chunk->last();
});

$chunks->all();

// [[['A', 'A'], ['B', 'B'], ['C', 'C', 'C']], ['D']]
```

## collapse()

Метод `collapse` сворачивает коллекцию массивов в единую плоскую коллекцию:

```
$collection = collect([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## collect()

Метод `collect` возвращает новый экземпляр `Collection` с элементами, находящимися в текущей коллекции:

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

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

Метод `collect` в первую очередь полезен для преобразования отложенных коллекций в стандартные экземпляры `Collection`:

```
$lazyCollection = LazyCollection::make(function () {
    yield 1;
    yield 2;
    yield 3;
});

$collection = $lazyCollection->collect();

$collection::class;

// 'Illuminate\Support\Collection'

$collection->all();

// [1, 2, 3]
```

Метод `collect` особенно полезен, когда у вас есть экземпляр `Enumerable` и вам нужен «не-отложенный» экземпляр коллекции. Так как `collect()` является частью контракта `Enumerable`, вы можете безопасно использовать его для получения экземпляра `Collection`.

## combine()

Метод `combine` объединяет значения коллекции в качестве ключей со значениями другого массива или коллекции:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

## concat()

Метод `concat` добавляет значения переданного массива или коллекции в конец другой коллекции:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```



Метод `concat` численно переиндексирует ключи для элементов, добавленных к исходной коллекции. Чтобы сохранить ключи в ассоциативных коллекциях, см. метод [merge](#).

## contains()

Метод `contains` определяет, содержит ли коллекция данный элемент. Вы можете передать в `contains` функцию, чтобы определить, существует ли в коллекции элемент, соответствующий указанному критерию истинности:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function (int $value, int $key) {
    return $value > 5;
});

// false
```

Вы также можете передать строку методу `contains`, чтобы определить, содержит ли коллекция указанное значение элемента:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');
```

```
// false
```

Вы также можете передать пару ключ / значение методу `contains`, который определит, существует ли данная пара в коллекции:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Метод `contains` использует «гибкое» сравнение при проверке значений элементов, то есть строка с целочисленным значением будет считаться равной целому числу того же значения. Используйте метод `containsStrict` для фильтрации с использованием «жесткого» сравнения.

Противоположным для метода `contains`, является метод `doesntContain`.

## containsOneItem()

Метод `containsOneItem` определяет, содержит ли коллекция только один элемент:

```
collect([])->containsOneItem();

// false

collect(['1'])->containsOneItem();

// true

collect(['1', '2'])->containsOneItem();

// false
```

## containsStrict()

Этот метод имеет ту же сигнатуру, что и метод [contains](#); однако, все значения сравниваются с использованием «жесткого» сравнения.

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## count()

Метод `count` возвращает общее количество элементов в коллекции:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->count();  
  
// 4
```

## countBy()

Метод `countBy` подсчитывает вхождения значений в коллекцию. По умолчанию метод подсчитывает вхождения каждого элемента, что позволяет подсчитать определенные «типы» элементов в коллекции:

```
$collection = collect([1, 2, 2, 2, 3]);  
  
$counted = $collection->countBy();  
  
$counted->all();  
  
// [1 => 1, 2 => 3, 3 => 1]
```

Вы можете передать замыкание методу `countBy` для подсчета всех элементов по собственным критериям:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);  
  
$counted = $collection->countBy(function (string $email) {  
    return substr(strrchr($email, "@"), 1);  
});
```

```
$counted->all();  
  
// ['gmail.com' => 2, 'yahoo.com' => 1]
```

## crossJoin()

Метод `crossJoin` перекрестно соединяет значения коллекции среди переданных массивов или коллекций, возвращая декартово произведение со всеми возможными перестановками:

```
$collection = collect([1, 2]);  
  
$matrix = $collection->crossJoin(['a', 'b']);  
  
$matrix->all();  
  
/*  
 [  
     [1, 'a'],  
     [1, 'b'],  
     [2, 'a'],  
     [2, 'b'],  
 ]  
 */  
  
$collection = collect([1, 2]);  
  
$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);  
  
$matrix->all();  
  
/*  
 [  
     [1, 'a', 'I'],  
     [1, 'a', 'II'],  
     [1, 'b', 'I'],  
     [1, 'b', 'II'],  
     [2, 'a', 'I'],  
     [2, 'a', 'II'],  
     [2, 'b', 'I'],  
     [2, 'b', 'II'],  
 ]  
 */
```

## dd()

Метод `dd` выводит элементы коллекции и завершает выполнение скрипта:

```
$collection = collect(['John Doe', 'Jane Doe']);  
  
$collection->dd();  
  
/*  
Collection {  
    #items: array:2 [  
        0 => "John Doe"  
        1 => "Jane Doe"  
    ]  
}  
*/
```

Если вы не хотите останавливать выполнение вашего скрипта, используйте вместо этого метод `dump`.

## diff()

Метод `diff` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его значений. Этот метод вернет значения из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$diff = $collection->diff([2, 4, 6, 8]);  
  
$diff->all();  
  
// [1, 3, 5]
```

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## diffAssoc()

Метод `diffAssoc` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его ключей и значений. Этот метод вернет пары ключ / значение из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

## diffAssocUsing()

В отличие от `diffAssoc`, `diffAssocUsing` принимает пользовательскую функцию обратного вызова для сравнения индексов:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssocUsing([
    'Color' => 'yellow',
    'Type' => 'fruit',
    'Remain' => 3,
], 'strnatcasecmp');

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

Обратный вызов должен быть функцией сравнения, которая возвращает целое число меньше, равное или больше нуля. Дополнительную информацию можно

найти в документации PHP по array\_diff\_uassoc, функции PHP, которую внутренне использует метод diffAssocUsing.

## diffKeys()

Метод `diffKeys` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его ключей. Этот метод вернет пары ключ / значение из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

## doesntContain()

Метод `doesntContain` определяет, не содержит ли коллекция данный элемент. Вы можете передать замыкание методу `doesntContain`, чтобы определить, не существует ли элемента в коллекции, соответствующего заданному критерию:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->doesntContain(function (int $value, int $key) {
    return $value < 5;
});

// false
```

В качестве альтернативы вы можете передать строку методу `doesntContain`, чтобы определить, не содержит ли коллекция заданного значения элемента:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);  
  
$collection->doesntContain('Table');  
  
// true  
  
$collection->doesntContain('Desk');  
  
// false
```

Вы также можете передать пару ключ/значение методу `doesntContain`, который определит, не существует ли данная пара в коллекции:

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],  
    ['product' => 'Chair', 'price' => 100],  
]);  
  
$collection->doesntContain('product', 'Bookcase');  
  
// true
```

Метод `doesntContain` использует “не строгое” сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения.

## dot()

Метод `dot` сглаживает многомерную коллекцию в коллекцию одного уровня, используя “точечную” нотацию для указания глубины:

```
$collection = collect(['products' => ['desk' => ['price' => 100]]]);  
  
$flattened = $collection->dot();  
  
$flattened->all();  
  
// ['products.desk.price' => 100]
```

## dump()

Метод `dump` выводит элементы коллекции:

```
$collection = collect(['John Doe', 'Jane Doe']);  
  
$collection->dump();  
  
/*  
 Collection {  
     #items: array:2 [  
         0 => "John Doe"  
         1 => "Jane Doe"  
     ]  
 }  
 */
```

Если вы хотите прекратить выполнение скрипта после вывода элементов коллекции, используйте вместо этого метод `dd`.

## duplicates()

Метод `duplicates` извлекает и возвращает повторяющиеся значения из коллекции:

```
$collection = collect(['a', 'b', 'a', 'c', 'b']);  
  
$collection->duplicates();  
  
// [2 => 'a', 4 => 'b']
```

Если коллекция содержит массивы или объекты, вы можете передать ключ атрибутов, которые вы хотите проверить на наличие повторяющихся значений:

```
$employees = collect([  
    ['email' => 'abigail@example.com', 'position' => 'Developer'],  
    ['email' => 'james@example.com', 'position' => 'Designer'],  
    ['email' => 'victoria@example.com', 'position' => 'Developer'],  
]);  
  
$employees->duplicates('position');  
  
// [2 => 'Developer']
```

## duplicatesStrict()

Этот метод имеет ту же сигнатуру, что и метод [duplicates](#); однако, все значения сравниваются с использованием «жесткого» сравнения.

## each()

Метод [each](#) перебирает элементы в коллекции и передает каждый элемент в замыкание:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->each(function (int $item, int $key) {  
    // ...  
});
```

Если вы хотите прекратить итерацию по элементам, вы можете вернуть [false](#) из вашего замыкания:

```
$collection->each(function (int $item, int $key) {  
    if /* condition */ {  
        return false;  
    }
```

## eachSpread()

Метод [eachSpread](#) выполняет итерацию по элементам коллекции, передавая значение каждого вложенного элемента в замыкание:

```
$collection = collect(['John Doe', 35], ['Jane Doe', 33]);  
  
$collection->eachSpread(function (string $name, int $age) {  
    // ...  
});
```

Если вы хотите прекратить итерацию по элементам, вы можете вернуть [false](#) из вашего замыкания:

```
$collection->eachSpread(function (string $name, int $age) {
    return false;
});
```

## ensure()

Метод `ensure` может использоваться для проверки того, что все элементы коллекции имеют определенный тип или список типов. В противном случае будет выброшено исключение `UnexpectedValueException`:

```
return $collection->ensure(User::class);

return $collection->ensure([User::class, Customer::class]);
```

Примитивные типы, такие как `string`, `int`, `float`, `bool` и `array`, также могут быть указаны:

```
return $collection->ensure('int');
```

Метод `ensure` не гарантирует, что элементы разных типов не будут добавлены в коллекцию в будущем.

## every()

Метод `every` используется для проверки того, что все элементы коллекции проходят указанный тест истинности:

```
collect([1, 2, 3, 4])->every(function (int $value, int $key) {
    return $value > 2;
});

// false
```

Если коллекция пуста, метод `every` вернет `true`:

```
$collection = collect([]);

$collection->every(function (int $value, int $key) {
    return $value > 2;
});

// true
```

## except()

Метод `except` возвращает все элементы из коллекции, кроме тех, которые имеют указанные ключи:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

Противоположным методу `except` является метод `only`.

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## filter()

Метод `filter` фильтрует коллекцию, используя переданное замыкание, сохраняя только те элементы, которые проходят указанный тест истинности:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function (int $value, int $key) {
    return $value > 2;
});

$filtered->all();
```

```
// [3, 4]
```

Если замыкание не указано, то все записи коллекции, эквивалентные `false`, будут удалены:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);  
  
$collection->filter()->all();  
  
// [1, 2, 3]
```

Противоположным методу `filter` является метод `reject`.

## first()

Метод `first` возвращает первый элемент из коллекции, который проходит указанную проверку истинности:

```
collect([1, 2, 3, 4])->first(function (int $value, int $key) {  
    return $value > 2;  
});  
  
// 3
```

Вы также можете вызвать метод `first` без аргументов, чтобы получить первый элемент из коллекции. Если коллекция пуста, возвращается `null`:

```
collect([1, 2, 3, 4])->first();  
  
// 1
```

## firstOrFail()

Метод `firstOrFail` идентичен методу `first`; однако, если результат не найден, будет сгенерировано исключение `Illuminate\Support\ItemNotFoundException`:

```
collect([1, 2, 3, 4])->firstOrFail(function (int $value, int $key) {  
    return $value > 5;
```

```
});  
  
// Генерирует исключение ItemNotFoundException...
```

Вы также можете вызвать метод `firstOrFail` без аргументов, чтобы получить первый элемент в коллекции. Если коллекция пуста, будет сгенерировано исключение `Illuminate\Support\ItemNotFoundException`:

```
collect([])->firstOrFail();  
  
// Генерирует исключение ItemNotFoundException...
```

## firstWhere()

Метод `firstWhere` возвращает первый элемент коллекции с переданной парой ключ / значение:

```
$collection = collect([  
    ['name' => 'Regena', 'age' => null],  
    ['name' => 'Linda', 'age' => 14],  
    ['name' => 'Diego', 'age' => 23],  
    ['name' => 'Linda', 'age' => 84],  
]);  
  
$collection->firstWhere('name', 'Linda');  
  
// ['name' => 'Linda', 'age' => 14]
```

Вы также можете вызвать метод `firstWhere` с оператором сравнения:

```
$collection->firstWhere('age', '>=', 18);  
  
// ['name' => 'Diego', 'age' => 23]
```

Подобно методу `where`, вы можете передать один аргумент методу `firstWhere`. В этом сценарии метод `firstWhere` вернет первый элемент, для которого значение данного ключа элемента является «истинным»:

```
$collection->firstWhere('age');
```

```
// ['name' => 'Linda', 'age' => 14]
```

## flatMap()

Метод `flatMap` выполняет итерацию по коллекции и передает каждое значение переданному замыканию. Замыкание может изменить элемент и вернуть его, таким образом формируя новую коллекцию измененных элементов. Затем массив преобразуется в плоскую структуру:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function (array $values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

## flatten()

Метод `flatten` объединяет многомерную коллекцию в одноуровневую:

```
$collection = collect([
    'name' => 'taylor',
    'languages' => [
        'php', 'javascript'
    ]
]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

Если необходимо, вы можете передать методу `flatten` аргумент «глубины»:

```

$collection = collect([
    'Apple' => [
        [
            'name' => 'iPhone 6S',
            'brand' => 'Apple'
        ],
    ],
    'Samsung' => [
        [
            'name' => 'Galaxy S7',
            'brand' => 'Samsung'
        ],
    ],
]);

```

```

$products = $collection->flatten(1);

$products->values()->all();

/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
*/

```

В этом примере вызов `flatten` без указания глубины также привел бы к сглаживанию вложенных массивов, что привело бы к `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Предоставление глубины позволяет указать количество уровней, на которые будут сглажены вложенные массивы.

## flip()

Метод `flip` меняет местами ключи коллекции на их соответствующие значения:

```

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']

```

## forget()

Метод `forget` удаляет элемент из коллекции по его ключу:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
// Забыть один ключ...  
$collection->forget('name');  
  
// ['framework' => 'laravel']  
  
// Забыть несколько ключей...  
$collection->forget(['name', 'framework']);  
  
// []
```

В отличие от большинства других методов коллекции, `forget` модифицирует коллекцию.

## forPage()

Метод `forPage` возвращает новую коллекцию, содержащую элементы, которые будут присутствовать на указанном номере страницы. Метод принимает номер страницы в качестве первого аргумента и количество элементов, отображаемых на странице, в качестве второго аргумента:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);  
  
$chunk = $collection->forPage(2, 3);  
  
$chunk->all();  
  
// [4, 5, 6]
```

## get()

Метод `get` возвращает элемент по указанному ключу. Если ключ не существует, возвращается `null`:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('name');  
  
// taylor
```

При желании вы можете передать значение по умолчанию в качестве второго аргумента:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('age', 34);  
  
// 34
```

Вы даже можете передать замыкание как значение метода по умолчанию. Результат замыкания будет возвращен, если указанный ключ не существует:

```
$collection->get('email', function () {  
    return 'taylor@example.com';  
});  
  
// taylor@example.com
```

## groupBy()

Метод `groupBy` группирует элементы коллекции по указанному ключу:

```
$collection = collect([  
    ['account_id' => 'account-x10', 'product' => 'Chair'],  
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],  
    ['account_id' => 'account-x11', 'product' => 'Desk'],  
]);  
  
$grouped = $collection->groupBy('account_id');  
  
$grouped->all();  
  
/*  
 [  
     'account-x10' => [  
         ['account_id' => 'account-x10', 'product' => 'Chair'],  
         ['account_id' => 'account-x10', 'product' => 'Bookcase'],  
     ],  
     'account-x11' => [  
         ['account_id' => 'account-x11', 'product' => 'Desk'],  
     ],  
 ]*/
```

```

        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

Вместо передачи строкового ключа вы можете передать замыкание. Замыкание должно вернуть значение, используемое в качестве ключа для группировки:

```

$grouped = $collection->groupBy(function (array $item, int $key) {
    return substr($item['account_id'], -3);
});

$grouped->all();

/*
[
    'x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

В виде массива можно передать несколько критериев группировки. Каждый элемент массива будет применен к соответствующему уровню в многомерном массиве:

```

$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy(['skill', function (array $item) {
    return $item['roles'];
}], preserveKeys: true);

/*

```

```
[  
 1 => [  
    'Role_1' => [  
      10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],  
      20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],  
    ],  
    'Role_2' => [  
      20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],  
    ],  
    'Role_3' => [  
      10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],  
    ],  
  ],  
 2 => [  
    'Role_1' => [  
      30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],  
    ],  
    'Role_2' => [  
      40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],  
    ],  
  ],  
];  
*/
```

## has()

Метод `has` определяет, существует ли переданный ключ в коллекции:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);  
  
$collection->has('product');  
  
// true  
  
$collection->has(['product', 'amount']);  
  
// true  
  
$collection->has(['amount', 'price']);  
  
// false
```

## hasAny()

Метод `hasAny` определяет, существует ли хотя бы один из заданных ключей в коллекции:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);  
  
$collection->hasAny(['product', 'price']);  
  
// true  
  
$collection->hasAny(['name', 'price']);  
  
// false
```

## implode()

Метод `implode` объединяет элементы коллекции. Его аргументы зависят от типа элементов в коллекции. Если коллекция содержит массивы или объекты, вы должны передать ключ объединяемых атрибутов, и «связующую строку», размещаемую между значениями:

```
$collection = collect([  
    ['account_id' => 1, 'product' => 'Desk'],  
    ['account_id' => 2, 'product' => 'Chair'],  
]);  
  
$collection->implode('product', ', ', );  
  
// Desk, Chair
```

Если коллекция содержит простые строки или числовые значения, вы должны передать «связующую строку» как единственный аргумент методу:

```
collect([1, 2, 3, 4, 5])->implode('-');  
  
// '1-2-3-4-5'
```

Вы можете передать замыкание методу `implode`, если хотите форматировать значения, которые объединяются:

```
$collection->implode(function (array $item, int $key) {  
    return strtoupper($item['product']);
```

```
}, ', ');  
// DESK, CHAIR
```

## intersect()

Метод `intersect` удаляет любые значения из исходной коллекции, которых нет в указанном массиве или коллекции. Полученная коллекция сохранит ключи исходной коллекции:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);  
  
$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);  
  
$intersect->all();  
  
// [0 => 'Desk', 2 => 'Chair']
```

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## intersectAssoc()

Метод `intersectAssoc` сравнивает исходную коллекцию с другой коллекцией или `array`, возвращая пары ключ / значение, которые присутствуют во всех заданных коллекциях:

```
$collection = collect([  
    'color' => 'red',  
    'size' => 'M',  
    'material' => 'cotton'  
]);  
  
$intersect = $collection->intersectAssoc([  
    'color' => 'blue',  
    'size' => 'M',  

```

```
$intersect->all();
```

```
// ['size' => 'M']
```

## intersectByKeys()

Метод `intersectByKeys` удаляет все ключи и соответствующие им значения из исходной коллекции, ключи которых отсутствуют в указанном массиве или коллекции:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009,
]);
```

```
$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011,
]);
```

```
$intersect->all();
```

```
// ['type' => 'screen', 'year' => 2009]
```

## isEmpty()

Метод `isEmpty` возвращает `true`, если коллекция пуста; в противном случае возвращается `false`:

```
collect([])->isEmpty();
```

```
// true
```

## isNotEmpty()

Метод `isNotEmpty` возвращает `true`, если коллекция не пуста; в противном случае возвращается `false`:

```
collect([])->isNotEmpty();
```

```
// false
```

## join()

Метод `join` объединяет значения коллекции в строку. Используя второй аргумент этого метода, вы также можете указать, как последний элемент должен быть добавлен к строке:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'  
collect(['a', 'b', 'c'])->join(', ', ' and '); // 'a, b, and c'  
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'  
collect(['a'])->join(', ', ' and '); // 'a'  
collect([])->join(', ', ' and '); // ''
```

## keyBy()

Метод `keyBy` возвращает коллекцию, элементы которой будут образованы путем присвоения ключей элементам базовой коллекции. Если у нескольких элементов один и тот же ключ, в новой коллекции появится только последний:

```
$collection = collect([  
    ['product_id' => 'prod-100', 'name' => 'Desk'],  
    ['product_id' => 'prod-200', 'name' => 'Chair'],  
]);  
  
$keyed = $collection->keyBy('product_id');  
  
$keyed->all();  
  
/*  
 [  
     'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],  
     'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],  
 ]  
 */
```

Вы также можете передать методу замыкание. Замыкание должно возвращать имя для ключа коллекции:

```
$keyed = $collection->keyBy(function (array $item, int $key) {  
    return strtoupper($item['product_id']);  
});  
  
$keyed->all();
```

```
/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

## keys()

Метод `keys` возвращает все ключи коллекции:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

## last()

Метод `last` возвращает последний элемент в коллекции, который проходит указанную проверку истинности:

```
collect([1, 2, 3, 4])->last(function (int $value, int $key) {
    return $value < 3;
});

// 2
```

Вы также можете вызвать метод `last` без аргументов, чтобы получить последний элемент коллекции. Если коллекция пуста, возвращается `null`:

```
collect([1, 2, 3, 4])->last();

// 4
```

## lazy()

Метод `lazy` возвращает новый экземпляр `LazyCollection` из базового массива элементов:

```
$lazyCollection = collect([1, 2, 3, 4])->lazy();  
  
$lazyCollection::class;  
  
// Illuminate\Support\LazyCollection  
  
$lazyCollection->all();  
  
// [1, 2, 3, 4]
```

Это особенно полезно, когда вам нужно выполнять преобразования в огромной `Collection`, содержащей множество элементов:

```
$count = $hugeCollection  
->lazy()  
->where('country', 'FR')  
->where('balance', '>', '100')  
->count();
```

Преобразовав коллекцию в `LazyCollection`, мы избегаем необходимости выделять огромное количество дополнительной памяти. Хотя исходная коллекция все еще хранит свои значения в памяти, последующие фильтры этого не делают. Таким образом, при фильтрации результатов коллекции практически не выделяется дополнительной памяти.

## macro()

Статический метод `macro` позволяет вам добавлять методы к классу `Collection` во время выполнения. Обратитесь к документации по [расширению коллекций](#) для получения дополнительной информации.

## make()

Статический метод `make` создает новый экземпляр коллекции. См. раздел [Создание коллекций](#).

## map()

Метод `map` выполняет итерацию по коллекции и передает каждое значение указанному замыканию. Замыкание может изменить элемент и вернуть его, образуя новую коллекцию измененных элементов:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$multiplied = $collection->map(function (int $item, int $key) {  
    return $item * 2;  
});  
  
$multiplied->all();  
  
// [2, 4, 6, 8, 10]
```

Как и большинство других методов коллекции, `map` возвращает новый экземпляр коллекции; он не модифицирует коллекцию. Если вы хотите преобразовать исходную коллекцию, используйте метод `transform`.

## mapInto()

Метод `mapInto()` выполняет итерацию коллекции, создавая новый экземпляр указанного класса, и передавая значение в его конструктор:

```
class Currency  
{  
    /**  
     * Создать новый экземпляр валюты.  
     */  
    function __construct(  
        public string $code  
    ) {}  
  
    $collection = collect(['USD', 'EUR', 'GBP']);  
  
    $currencies = $collection->mapInto(Currency::class);
```

```
$currencies->all();  
  
// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

## mapSpread()

Метод `mapSpread` выполняет итерацию по элементам коллекции, передавая значение каждого вложенного элемента в указанное замыкание. Замыкание может изменить элемент и вернуть его, таким образом формируя новую коллекцию измененных элементов:

```
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);  
  
$chunks = $collection->chunk(2);  
  
$sequence = $chunks->mapSpread(function (int $even, int $odd)  
    return $even + $odd;  
});  
  
$sequence->all();  
  
// [1, 5, 9, 13, 17]
```

## mapToGroups()

Метод `mapToGroups` группирует элементы коллекции по указанному замыканию. Замыкание должно возвращать ассоциативный массив, содержащий одну пару ключ / значение, таким образом формируя новую коллекцию сгруппированных значений:

```
$collection = collect([  
    [  
        'name' => 'John Doe',  
        'department' => 'Sales',  
    ],  
    [  
        'name' => 'Jane Doe',  
        'department' => 'Sales',  
    ],  
    [  
        'name' => 'Johnny Doe',  
        'department' => 'Marketing',  
    ]
```

```

    ]
]);
```

```

$grouped = $collection->mapToGroups(function (array $item, int $key) {
    return [$item['department'] => $item['name']];
});
```

```

$grouped->all();
```

```

/*
[
    'Sales' => ['John Doe', 'Jane Doe'],
    'Marketing' => ['Johnny Doe'],
]
*/
```

```

$grouped->get('Sales')->all();
```

```

// ['John Doe', 'Jane Doe']
```

## mapWithKeys()

Метод `mapWithKeys` выполняет итерацию по коллекции и передает каждое значение в указанное замыкание. Замыкание должно возвращать ассоциативный массив, содержащий одну пару ключ / значение:

```

$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
]);
```

```

$keyed = $collection->mapWithKeys(function (array $item, int $key) {
    return [$item['email'] => $item['name']];
});
```

```

$keyed->all();
```

```

/*
[

```

```
'john@example.com' => 'John',
'jane@example.com' => 'Jane',
]
*/
*/
```

## max()

Метод `max` возвращает максимальное значение переданного ключа:

```
$max = collect([
    ['foo' => 10],
    ['foo' => 20]
])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

## median()

Метод `median` возвращает медиану переданного ключа:

```
$median = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

## merge()

Метод `merge` объединяет переданный массив или коллекцию с исходной коллекцией. Если строковый ключ в переданных элементах соответствует

строковому ключу в исходной коллекции, значение переданного элемента перезапишет значение в исходной коллекции:

```
$collection = collect(['product_id' => 1, 'price' => 100]);  
  
$merged = $collection->merge(['price' => 200, 'discount' => false]);  
  
$merged->all();  
  
// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

Если ключи переданных элементов являются числовыми, значения будут добавлены в конец коллекции:

```
$collection = collect(['Desk', 'Chair']);  
  
$merged = $collection->merge(['Bookcase', 'Door']);  
  
$merged->all();  
  
// ['Desk', 'Chair', 'Bookcase', 'Door']
```

## mergeRecursive()

Метод `mergeRecursive` рекурсивно объединяет переданный массив или коллекцию с исходной коллекцией. Если строковый ключ в переданных элементах совпадает со строковым ключом в исходной коллекции, тогда значения этих ключей объединяются в массив, и это делается рекурсивно:

```
$collection = collect(['product_id' => 1, 'price' => 100]);  
  
$merged = $collection->mergeRecursive([  
    'product_id' => 2,  
    'price' => 200,  
    'discount' => false  
]);  
  
$merged->all();  
  
// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

## min()

Метод `min` возвращает минимальное значение переданного ключа:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

## mode()

Метод `mode` возвращает значение моды указанного ключа:

```
$mode = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]

$mode = collect([1, 1, 2, 2])->mode();

// [1, 2]
```

## multiply()

Метод `multiply` создает указанное количество копий всех элементов коллекции:

```
$users = collect([
    ['name' => 'User #1', 'email' => 'user1@example.com'],
    ['name' => 'User #2', 'email' => 'user2@example.com'],
])->multiply(3);

/*
  [
    ['name' => 'User #1', 'email' => 'user1@example.com'],
    ['name' => 'User #1', 'email' => 'user1@example.com'],
    ['name' => 'User #1', 'email' => 'user1@example.com'],
    ['name' => 'User #2', 'email' => 'user2@example.com'],
    ['name' => 'User #2', 'email' => 'user2@example.com'],
    ['name' => 'User #2', 'email' => 'user2@example.com']
]
```

```
[  
    ['name' => 'User #1', 'email' => 'user1@example.com'],  
    ['name' => 'User #2', 'email' => 'user2@example.com'],  
    ['name' => 'User #1', 'email' => 'user1@example.com'],  
    ['name' => 'User #2', 'email' => 'user2@example.com'],  
    ['name' => 'User #1', 'email' => 'user1@example.com'],  
    ['name' => 'User #2', 'email' => 'user2@example.com'],  
]  
*/
```

## nth()

Метод `nth` создает новую коллекцию, состоящую из каждого `n`-го элемента:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);  
  
$collection->nth(4);  
  
// ['a', 'e']
```

При желании вы можете передать начальное смещение в качестве второго аргумента:

```
$collection->nth(4, 1);  
  
// ['b', 'f']
```

## only()

Метод `only` возвращает элементы коллекции только с указанными ключами:

```
$collection = collect([  
    'product_id' => 1,  
    'name' => 'Desk',  
    'price' => 100,  
    'discount' => false  
]);  
  
$filtered = $collection->only(['product_id', 'name']);  
  
$filtered->all();
```

```
// ['product_id' => 1, 'name' => 'Desk']
```

Противоположным методу `only` является метод `except`.

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## pad()

Метод `pad` дополнит коллекцию определенным значением, пока коллекция не достигнет указанного размера. Этот метод ведет себя как функция [array\\_pad](#) PHP.

Для дополнения слева следует указать отрицательный размер. Если абсолютное значение указанного размера меньше или равно длине массива, заполнение не произойдет:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

## partition()

Метод `partition` используется в связке с деструктуризацией массивов PHP (вместо функции `list` в предыдущих версиях), чтобы разделить элементы, прошедшие указанную проверку истинности, от тех, которые ее не прошли:

```
$collection = collect([1, 2, 3, 4, 5, 6]);  
  
[$underThree, $equalOrAboveThree] = $collection->partition(function (int $i) {  
    return $i < 3;  
});  
  
$underThree->all();  
  
// [1, 2]  
  
$equalOrAboveThree->all();  
  
// [3, 4, 5, 6]
```

## percentage()

Метод `percentage` может быть использован для быстрого определения процента элементов в коллекции, которые проходят заданное условие:

```
$collection = collect([1, 1, 2, 2, 2, 3]);  
  
$percentage = $collection->percentage(fn ($value) => $value === 1);  
  
// 33.33
```

По умолчанию процент будет округлен до двух знаков после запятой. Однако, вы можете настроить это поведение, указав второй аргумент метода:

```
$percentage = $collection->percentage(fn ($value) => $value === 1, precision: 3);  
  
// 33.333
```

## pipe()

Метод `pipe` передает коллекцию указанному замыканию и возвращает результат выполненного замыкания:

```
$collection = collect([1, 2, 3]);  
  
$piped = $collection->pipe(function (Collection $collection) {  
    return $collection->sum();
```

```
});
```

```
// 6
```

## pipeInto()

Метод `pipeInto` создает новый экземпляр указанного класса и передает коллекцию в конструктор:

```
class ResourceCollection
{
    /**
     * Создать новый экземпляр ResourceCollection.
     */
    public function __construct(
        public Collection $collection,
    ) {}
}

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

$resource = $collection->pipeInto(ResourceCollection::class);

$resource->collection->all();

// [1, 2, 3]
```

## pipeThrough()

Метод `pipeThrough` передает коллекцию заданному массиву замыканий и возвращает результат выполненных замыканий:

```
use Illuminate\Support\Collection;

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

$result = $collection->pipeThrough([
    function (Collection $collection) {
        return $collection->merge([4, 5]);
    },
    function (Collection $collection) {
        return $collection->sum();
    },
]);
```

```
]);
```

```
// 15
```

## pluck()

Метод `pluck` извлекает все значения для указанного ключа:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

Вы также можете задать ключ результирующей коллекции:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

Метод `pluck` также поддерживает получение вложенных значений с использованием «точечной нотации»:

```
$collection = collect([
    [
        'name' => 'Laracon',
        'speakers' => [
            'first_day' => ['Rosa', 'Judith'],
        ],
    ],
    [
        'name' => 'VueConf',
        'speakers' => [
            'first_day' => ['Abigail', 'Joey'],
        ],
    ],
],
```

```
]);  
  
$plucked = $collection->pluck('speakers.first_day');  
  
$plucked->all();  
  
// [['Rosa', 'Judith'], ['Abigail', 'Joey']]
```

Если существуют повторяющиеся ключи, последний соответствующий элемент будет вставлен в результирующую коллекцию:

```
$collection = collect([  
    ['brand' => 'Tesla', 'color' => 'red'],  
    ['brand' => 'Pagani', 'color' => 'white'],  
    ['brand' => 'Tesla', 'color' => 'black'],  
    ['brand' => 'Pagani', 'color' => 'orange'],  
]);  
  
$plucked = $collection->pluck('color', 'brand');  
  
$plucked->all();  
  
// ['Tesla' => 'black', 'Pagani' => 'orange']
```

## pop()

Метод `pop` удаляет и возвращает последний элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->pop();  
  
// 5  
  
$collection->all();  
  
// [1, 2, 3, 4]
```

Вы можете передать целое число в метод `pop`, чтобы удалить и вернуть несколько элементов из конца коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->pop(3);  
  
// collect([5, 4, 3])  
  
$collection->all();  
  
// [1, 2]
```

## prepend()

Метод `prepend` добавляет элемент в начало коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->prepend(0);  
  
$collection->all();  
  
// [0, 1, 2, 3, 4, 5]
```

Вы также можете передать второй аргумент, чтобы указать ключ добавляемого элемента:

```
$collection = collect(['one' => 1, 'two' => 2]);  
  
$collection->prepend(0, 'zero');  
  
$collection->all();  
  
// ['zero' => 0, 'one' => 1, 'two' => 2]
```

## pull()

Метод `pull` удаляет и возвращает элемент из коллекции по его ключу:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);  
  
$collection->pull('name');  
  
// 'Desk'
```

```
$collection->all();  
  
// ['product_id' => 'prod-100']
```

## push()

Метод `push` добавляет элемент в конец коллекции:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->push(5);  
  
$collection->all();  
  
// [1, 2, 3, 4, 5]
```

## put()

Метод `put` помещает указанные ключ и значение в коллекцию:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);  
  
$collection->put('price', 100);  
  
$collection->all();  
  
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

## random()

Метод `random` возвращает случайный элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->random();  
  
// 4 - (retrieved randomly)
```

Вы можете передать целое число в `random`, чтобы указать, сколько случайных элементов вы хотите получить. Коллекция элементов всегда возвращается при

явной передаче количества элементов, которые вы хотите получить:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

Если в экземпляре коллекции меньше элементов, чем запрошено, метод `random` сгенерирует исключение `InvalidArgumentException`.

Метод `random` также принимает замыкание, которое будет получать текущий экземпляр коллекции:

```
use Illuminate\Support\Collection;

$random = $collection->random(fn (Collection $items) => min(10, count($items)));

$random->all();

// [1, 2, 3, 4, 5] - (retrieved randomly)
```

## range()

Метод `range` возвращает коллекцию, содержащую целые числа в указанном диапазоне:

```
$collection = collect()->range(3, 6);

$collection->all();

// [3, 4, 5, 6]
```

## reduce()

Метод `reduce` сокращает коллекцию до одного значения, передавая результат каждой итерации следующей итерации:

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

```
$total = $collection->reduce(function (?int $carry, int $item) {
    return $carry + $item;
});

// 6
```

Значение `$carry` первой итерации равно `null`; однако вы можете указать его начальное значение, передав второй аргумент методу `reduce`:

```
$collection->reduce(function (int $carry, int $item) {
    return $carry + $item;
}, 4);

// 10
```

Метод `reduce` также передает ключи массива ассоциативных коллекций указанному замыканию:

```
$collection = collect([
    'usd' => 1400,
    'gbp' => 1200,
    'eur' => 1000,
]);

$ratio = [
    'usd' => 1,
    'gbp' => 1.37,
    'eur' => 1.22,
];

$collection->reduce(function (int $carry, int $value, int $key) use ($ratio) {
    return $carry + ($value * $ratio[$key]);
});

// 4264
```

## reduceSpread()

Метод `reduceSpread` сокращает коллекцию до массива значений, передавая результаты каждой итерации в следующую итерацию. Этот метод похож на метод `reduce`, однако он может принимать несколько начальных значений:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
->get()
->reduceSpread(function (int $creditsRemaining, Collection $batch, Image $image)
    if ($creditsRemaining >= $image->creditsRequired()) {
        $batch->push($image);

        $creditsRemaining -= $image->creditsRequired();
    }

    return [$creditsRemaining, $batch];
}, $creditsAvailable, collect());
```

## reject()

Метод `reject` фильтрует коллекцию, используя переданное замыкание. Замыкание должно возвращать `true`, если элемент должен быть удален из результирующей коллекции:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function (int $value, int $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

Противоположным методу `reject` является метод [filter](#).

## replace()

Метод `replace` ведет себя аналогично методу `merge`; однако, помимо перезаписи совпадающих элементов, имеющих строковые ключи, метод `replace` также перезаписывает элементы в коллекции, у которых есть совпадающие числовые ключи:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);
```

```
$replaced->all();  
  
// ['Taylor', 'Victoria', 'James', 'Finn']
```

## replaceRecursive()

Этот метод работает как и `replace`, но он будет повторяться в массивах и применять тот же процесс замены к внутренним значениям:

```
$collection = collect([  
    'Taylor',  
    'Abigail',  
    [  
        'James',  
        'Victoria',  
        'Finn'  
    ]  
]);  
  
$replaced = $collection->replaceRecursive([  
    'Charlie',  
    2 => [1 => 'King']  

```

## reverse()

Метод `reverse` меняет порядок элементов коллекции на обратный, сохраняя исходные ключи:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);  
  
$reversed = $collection->reverse();  
  
$reversed->all();  
  
/*  
 *  
 * [  
 *     4 => 'e',  
 *     3 => 'd',  
 *     2 => 'c',  
 *
```

```
    1 => 'b',
    0 => 'a',
]
*/
```

## search()

Метод `search` ищет в коллекции указанное значение и возвращает его ключ, если он найден. Если элемент не найден, возвращается `false`:

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

Поиск выполняется с использованием «гибкого» сравнения, то есть строка с целым значением будет считаться равной целому числу того же значения. Чтобы использовать «жесткое» сравнение, передайте `true` в качестве второго аргумента метода:

```
collect([2, 4, 6, 8])->search('4', strict: true);

// false
```

В качестве альтернативы вы можете передать собственное замыкание для поиска первого элемента, который проходит указанный тест на истинность:

```
collect([2, 4, 6, 8])->search(function (int $item, int $key) {
    return $item > 5;
});

// 2
```

## select() {.collection-method}

Метод `select` выбирает заданные ключи из коллекции, подобно SQL-оператору `SELECT`:

```
$users = collect([
    ['name' => 'Taylor Otwell', 'role' => 'Developer', 'status' => 'active'],
    ['name' => 'Victoria Faith', 'role' => 'Researcher', 'status' => 'active'],
]);

$users->select(['name', 'role']);

/*
[
    ['name' => 'Taylor Otwell', 'role' => 'Developer'],
    ['name' => 'Victoria Faith', 'role' => 'Researcher'],
],
*/
```

## shift()

Метод `shift` удаляет и возвращает первый элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

Вы можете передать целое число в метод `shift`, чтобы удалить и вернуть несколько элементов из начала коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift(3);

// collect([1, 2, 3])

$collection->all();

// [4, 5]
```

## shuffle()

Метод `shuffle` случайным образом перемешивает элементы в коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$shuffled = $collection->shuffle();  
  
$shuffled->all();  
  
// [3, 2, 5, 1, 4] - (последовательность случайная)
```

## skip()

Метод `skip` возвращает новую коллекцию с указанным количеством удаляемых из начала коллекции элементов:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
  
$collection = $collection->skip(4);  
  
$collection->all();  
  
// [5, 6, 7, 8, 9, 10]
```

## skipUntil()

Метод `skipUntil` пропускает элементы из коллекции до тех пор, пока переданное замыкание не вернет `true`, а затем вернет оставшиеся элементы в коллекции как новый экземпляр коллекции:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->skipUntil(function (int $item) {  
    return $item >= 3;  
});  
  
$subset->all();  
  
// [3, 4]
```

Вы также можете передать простое значение методу `skipUntil`, чтобы пропустить все элементы, пока не будет найдено указанное значение:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->skipUntil(3);  
  
$subset->all();  
  
// [3, 4]
```

Если указанное значение не найдено или замыкание никогда не возвращает `true`, то метод `skipUntil` вернет пустую коллекцию.

## skipWhile()

Метод `skipWhile` пропускает элементы из коллекции, пока указанное замыкание возвращает `true`, а затем возвращает оставшиеся элементы в коллекции как новую коллекцию:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->skipWhile(function (int $item) {  
    return $item <= 3;  
});  
  
$subset->all();  
  
// [4]
```

Если замыкание никогда не возвращает `false`, то метод `skipWhile` вернет пустую коллекцию.

## slice()

Метод `slice` возвращает фрагмент коллекции, начиная с указанного индекса:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
  
$slice = $collection->slice(4);  
  
$slice->all();  
  
// [5, 6, 7, 8, 9, 10]
```

Если вы хотите ограничить размер возвращаемого фрагмента, то передайте желаемый размер в качестве второго аргумента метода:

```
$slice = $collection->slice(4, 2);  
  
$slice->all();  
  
// [5, 6]
```

Возвращенный фрагмент по умолчанию сохранит ключи. Если вы не хотите сохранять исходные ключи, вы можете использовать метод [values](#), чтобы переиндексировать их.

## sliding()

Метод [sliding](#) возвращает новую коллекцию фрагментов (chunks), представляющих представление элементов коллекции в виде “скользящего окна”:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunks = $collection->sliding(2);  
  
$chunks->toArray();  
  
// [[1, 2], [2, 3], [3, 4], [4, 5]]
```

Это особенно полезно в сочетании с методом [eachSpread](#):

```
$transactions->sliding(2)->eachSpread(function (Collection $previous, Collection $current) {  
    $current->total = $previous->total + $current->amount;  
});
```



По желанию вторым аргументом можно передать "шаг", который определяет расстояние между первым элементом каждого фрагмента:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunks = $collection->sliding(3, step: 2);  
  
$chunks->toArray();  
  
// [[1, 2, 3], [3, 4, 5]]
```

## sole()

Метод `sole` возвращает первый элемент в коллекции, который проходит заданный тест на истинность, но только если тест на истинность соответствует ровно одному элементу:

```
collect([1, 2, 3, 4])->sole(function (int $value, int $key) {  
    return $value === 2;  
});  
  
// 2
```

Вы также можете передать пару ключ / значение в метод `sole`, который вернет первый элемент коллекции, соответствующий данной паре, но только в том случае, если совпадает ровно один элемент:

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],  
    ['product' => 'Chair', 'price' => 100],  
]);  
  
$collection->sole('product', 'Chair');  
  
// ['product' => 'Chair', 'price' => 100]
```

В качестве альтернативы вы также можете вызвать метод `sole` без аргумента, чтобы получить первый элемент в коллекции, если в ней только один элемент:

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],
```

```
]);  
  
$collection->sole();  
  
// ['product' => 'Desk', 'price' => 200]
```

Если в коллекции нет элементов, которые должны быть возвращены методом `sole`, будет брошено исключение `\Illuminate\Collections\ItemNotFoundException`. Если есть более одного элемента, который должен быть возвращен, то будет брошено исключение `\Illuminate\Collections\MultipleItemsFoundException`.

## some()

Псевдоним для метода [contains](#).

## sort()

Метод `sort` сортирует коллекцию. В отсортированной коллекции хранятся исходные ключи массива, поэтому в следующем примере мы будем использовать метод [values](#) для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([5, 3, 1, 2, 4]);  
  
$sorted = $collection->sort();  
  
$sorted->values()->all();  
  
// [1, 2, 3, 4, 5]
```

Если ваши потребности в сортировке более сложны, вы можете передать замыкание методу `sort` с вашим собственным алгоритмом. Обратитесь к документации PHP по [uasort](#), который используется внутри метода `sort`.

Если вам нужно отсортировать коллекцию вложенных массивов или объектов, то см. методы [sortBy](#) и [sortByDesc](#).

## sortBy()

Метод `sortBy` сортирует коллекцию по указанному ключу. В отсортированной коллекции хранятся исходные ключи массива, поэтому в следующем примере мы будем использовать метод `values` для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/
```

Метод `sortBy` принимает [флаги типа сортировки](#) в качестве второго аргумента:

```
$collection = collect([
    ['title' => 'Item 1'],
    ['title' => 'Item 12'],
    ['title' => 'Item 3'],
]);

$sorted = $collection->sortBy('title', SORT_NATURAL);

$sorted->values()->all();

/*
[
    ['title' => 'Item 1'],
    ['title' => 'Item 3'],
    ['title' => 'Item 12'],
]
*/
```

В качестве альтернативы вы можете передать собственное замыкание, чтобы определить, как сортировать значения коллекции:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function (array $product, int $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]
*/
```

Если вы хотите отсортировать свою коллекцию по нескольким атрибутам, вы можете передать массив операций сортировки методу `sortBy`. Каждая операция сортировки должна быть массивом, состоящим из атрибута, по которому вы хотите сортировать, и направления желаемой сортировки:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    ['name', 'asc'],
    ['age', 'desc'],
]);

$sorted->values()->all();

/*
[
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Taylor Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 36],
]
```

```

        ['name' => 'Abigail Otwell', 'age' => 30],
        ['name' => 'Taylor Otwell', 'age' => 36],
        ['name' => 'Taylor Otwell', 'age' => 34],
    ]
*/

```

При сортировке коллекции по нескольким атрибутам вы также можете указать замыкания, определяющие каждую операцию сортировки:

```

$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    fn (array $a, array $b) => $a['name'] <=> $b['name'],
    fn (array $a, array $b) => $b['age'] <=> $a['age'],
]);
$sorted->values()->all();

/*
[
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Taylor Otwell', 'age' => 34],
]
*/

```

## sortByDesc()

Этот метод имеет ту же сигнатуру, что и метод [sortBy](#), но отсортирует коллекцию в обратном порядке.

## sortDesc()

Этот метод сортирует коллекцию в порядке, обратном методу [sort](#):

```

$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sortDesc();

```

```
$sorted->values()->all();
```

```
// [5, 4, 3, 2, 1]
```

В отличие от `sort`, вы не можете передавать замыкание в `sortDesc`. Вместо этого вы должны использовать метод `sort` и инвертировать ваше сравнение.

## sortKeys()

Метод `sortKeys` сортирует коллекцию по ключам базового ассоциативного массива:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
[
    'first' => 'John',
    'id' => 22345,
    'last' => 'Doe',
]
*/
```

## sortKeysDesc()

Этот метод имеет ту же сигнатуру, что и метод `sortKeys`, но отсортирует коллекцию в обратном порядке.

## sortKeysUsing()

Метод `sortKeysUsing` сортирует коллекцию по ключам базового ассоциативного массива с помощью обратного вызова:

```
$collection = collect([
    'ID' => 22345,
```

```
'first' => 'John',
'last'  => 'Doe',
]);

$sorted = $collection->sortKeysUsing('strnatcasecmp');

$sorted->all();

/*
[
    'first' => 'John',
    'ID'   => 22345,
    'last'  => 'Doe',
]
*/
*/
```

Обратный вызов должен быть функцией сравнения, которая возвращает целое число, меньшее, равное или большее нуля. Для получения дополнительной информации обратитесь к документации по PHP [uksort](#), которая представляет собой функцию PHP, используемую внутри метода [sortKeysUsing](#).

## splice()

Метод [splice](#) удаляет и возвращает фрагмент элементов, начиная с указанного индекса:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

Вы можете передать второй аргумент, чтобы ограничить размер результирующей коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);
```

```
$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

Кроме того, вы можете передать третий аргумент, содержащий новые элементы, чтобы заменить элементы, удаленные из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

## split()

Метод `split` разбивает коллекцию на указанное количество групп:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->all();

// [[1, 2], [3, 4], [5]]
```

## splitIn()

Метод `splitIn` разбивает коллекцию на указанное количество групп, полностью заполняя нетерминальные группы перед тем, как выделить остаток последней группе:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
  
$groups = $collection->splitIn(3);  
  
$groups->all();  
  
// [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

## sum()

Метод `sum` возвращает сумму всех элементов в коллекции:

```
collect([1, 2, 3, 4, 5])->sum();  
  
// 15
```

Если коллекция содержит вложенные массивы или объекты, вы должны передать ключ, который будет использоваться для определения суммирования значений:

```
$collection = collect([  
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],  
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],  
]);  
  
$collection->sum('pages');  
  
// 1272
```

Кроме того, вы можете передать собственное замыкание, чтобы определить, какие значения коллекции суммировать:

```
$collection = collect([  
    ['name' => 'Chair', 'colors' => ['Black']],  
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],  
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],  
]);  
  
$collection->sum(function (array $product) {  
    return count($product['colors']);  
});
```

```
// 6
```

## take()

Метод `take` возвращает новую коллекцию с указанным количеством элементов:

```
$collection = collect([0, 1, 2, 3, 4, 5]);  
  
$chunk = $collection->take(3);  
  
$chunk->all();  
  
// [0, 1, 2]
```

Вы также можете передать отрицательное целое число, чтобы получить указанное количество элементов из конца коллекции:

```
$collection = collect([0, 1, 2, 3, 4, 5]);  
  
$chunk = $collection->take(-2);  
  
$chunk->all();  
  
// [4, 5]
```

## takeUntil()

Метод `takeUntil` возвращает элементы коллекции, пока указанное замыкание не вернет `true`:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->takeUntil(function (int $item) {  
    return $item >= 3;  
});  
  
$subset->all();  
  
// [1, 2]
```

Вы также можете передать простое значение методу `takeUntil`, чтобы получать элементы, пока не будет найдено указанное значение:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->takeUntil(3);  
  
$subset->all();  
  
// [1, 2]
```

Если указанное значение не найдено или замыкание никогда не возвращает `true`, то метод `takeUntil` вернет все элементы коллекции.

## takeWhile()

Метод `takeWhile` возвращает элементы коллекции до тех пор, пока указанное замыкание не вернет `false`:

```
$collection = collect([1, 2, 3, 4]);  
  
$subset = $collection->takeWhile(function (int $item) {  
    return $item < 3;  
});  
  
$subset->all();  
  
// [1, 2]
```

Если замыкание никогда не возвращает `false`, метод `takeWhile` вернет все элементы коллекции.

## tap()

Метод `tap` передает коллекцию указанному замыканию, позволяя вам «перехватить» коллекцию в определенный момент и сделать что-то с элементами, не затрагивая саму коллекцию. Затем коллекция возвращается методом `tap`:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function (Collection $collection) {
        Log::debug('Values after sorting', $collection->values()->all());
    })
    ->shift();

// 1
```

## times()

Статический метод `times` создает новую коллекцию, вызывая переданное замыкание указанное количество раз:

```
$collection = Collection::times(10, function (int $number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

## toArray()

Метод `toArray` преобразует коллекцию в простой массив PHP. Если значениями коллекции являются модели [Eloquent](#), то модели также будут преобразованы в массивы:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
[
    ['name' => 'Desk', 'price' => 200],
]
*/
```

Метод `toArray` также преобразует все вложенные объекты коллекции, которые являются экземпляром `Arrayable`, в массив. Если вы хотите получить необработанный массив, лежащий в основе коллекции, используйте вместо этого метод `all`.

## toJson()

Метод `toJson` преобразует коллекцию в сериализованную строку JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toJson();  
  
// '{"name": "Desk", "price": 200}'
```

## transform()

Метод `transform` выполняет итерацию коллекции и вызывает указанное замыкание для каждого элемента в коллекции. Элементы в коллекции будут заменены значениями, возвращаемыми замыканием:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->transform(function (int $item, int $key) {  
    return $item * 2;  
});  
  
$collection->all();  
  
// [2, 4, 6, 8, 10]
```

В отличие от большинства других методов коллекции, `transform` модифицирует коллекцию. Если

вы хотите вместо этого создать новую коллекцию, используйте метод [map](#).

## undot()

Метод [undot](#) расширяет одномерную коллекцию, использующую «точечную» нотацию, в многомерную коллекцию:

```
$person = collect([
    'name.first_name' => 'Marie',
    'name.last_name' => 'Valentine',
    'address.line_1' => '2992 Eagle Drive',
    'address.line_2' => '',
    'address.suburb' => 'Detroit',
    'address.state' => 'MI',
    'address.postcode' => '48219'
]);

$person = $person->undot();

$person->toArray();

/*
[
    "name" => [
        "first_name" => "Marie",
        "last_name" => "Valentine",
    ],
    "address" => [
        "line_1" => "2992 Eagle Drive",
        "line_2" => "",
        "suburb" => "Detroit",
        "state" => "MI",
        "postcode" => "48219",
    ],
]
*/
```

## union()

Метод [union](#) добавляет переданный массив в коллекцию. Если переданный массив содержит ключи, которые уже находятся в исходной коллекции, предпочтительнее будут значения исходной коллекции:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['d']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

## unique()

Метод `unique` возвращает все уникальные элементы коллекции. Возвращенная коллекция сохраняет исходные ключи массива, поэтому в следующем примере мы будем использовать метод `values` для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

При работе с вложенными массивами или объектами вы можете указать ключ, используемый для определения уникальности:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
]
```

```
]  
*/
```

Наконец, вы также можете передать собственное замыкание методу `unique`, чтобы указать, какое значение должно определять уникальность элемента:

```
$unique = $collection->unique(function (array $item) {  
    return $item['brand'].$item['type'];  
});  
  
$unique->values()->all();  
  
/*  
[  
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],  
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],  
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],  
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],  
]  
*/
```

Метод `unique` использует «гибкое» сравнение при проверке значений элементов, то есть строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `uniqueStrict` для фильтрации с использованием «жесткого» сравнения.

Поведение этого метода изменяется при использовании [коллекций Eloquent](#).

## uniqueStrict()

Этот метод имеет ту же сигнатуру, что и метод `unique`; однако, все значения сравниваются с использованием «жесткого» сравнения.

## unless()

Метод `unless` выполнит указанное замыкание, если первый аргумент, переданный методу, не будет иметь значение `true`:

```
$collection = collect([1, 2, 3]);  
  
$collection->unless(true, function (Collection $collection) {  
    return $collection->push(4);  
});  
  
$collection->unless(false, function (Collection $collection) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 5]
```

Второе замыкание может быть передано методу `unless`. Второе замыкание будет выполнено, когда первый аргумент, переданный методу `unless` будет иметь значение `true`:

```
$collection = collect([1, 2, 3]);  
  
$collection->unless(true, function (Collection $collection) {  
    return $collection->push(4);  
}, function (Collection $collection) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 5]
```

Противоположным методу `unless` является метод `when`.

## unlessEmpty()

Псевдоним для метода `whenNotEmpty`.

## unlessNotEmpty()

Псевдоним для метода `whenEmpty`.

## unwrap()

Статический метод `unwrap` возвращает базовые элементы коллекции из указанного значения, когда это применимо:

```
Collection::unwrap(collect('John Doe'));  
  
// ['John Doe']  
  
Collection::unwrap(['John Doe']);  
  
// ['John Doe']  
  
Collection::unwrap('John Doe');  
  
// 'John Doe'
```

## value()

Метод `value` извлекает заданное значение из первого элемента коллекции:

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],  
    ['product' => 'Speaker', 'price' => 400],  
]);  
  
$value = $collection->value('price');  
  
// 200
```

## values()

Метод `values` возвращает новую коллекцию с ключами, сброшенными на последовательные целые числа:

```
$collection = collect([  
    10 => ['product' => 'Desk', 'price' => 200],  
    11 => ['product' => 'Desk', 'price' => 200],  
]);  
  
$values = $collection->values();  
  
$values->all();  
  
/*
```

```
[  
    0 => ['product' => 'Desk', 'price' => 200],  
    1 => ['product' => 'Desk', 'price' => 200],  
]  
*/
```

## when()

Метод `when` выполнит указанное замыкание, когда первый аргумент, переданный методу, оценивается как `true`. Экземпляр коллекции и первый аргумент, переданный методу `when`, будут предоставлены в замыкание:

```
$collection = collect([1, 2, 3]);  
  
$collection->when(true, function (Collection $collection, int $value) {  
    return $collection->push(4);  
});  
  
$collection->when(false, function (Collection $collection, int $value) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 4]
```

Второе замыкание может быть передано методу `when`. Второе замыкание будет выполнено, когда первый аргумент, переданный методу `when` будет иметь значение `false`:

```
$collection = collect([1, 2, 3]);  
  
$collection->when(false, function (Collection $collection, int $value) {  
    return $collection->push(4);  
}, function (Collection $collection) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 5]
```

Противоположным методу `when` является метод `unless`.

## whenEmpty()

Метод `whenEmpty` выполнит указанное замыкание, когда коллекция пуста:

```
$collection = collect(['Michael', 'Tom']);  
  
$collection->whenEmpty(function (Collection $collection) {  
    return $collection->push('Adam');  
});  
  
$collection->all();  
  
// ['Michael', 'Tom']  
  
  
$collection = collect();  
  
$collection->whenEmpty(function (Collection $collection) {  
    return $collection->push('Adam');  
});  
  
$collection->all();  
  
// ['Adam']
```

Второе замыкание может быть передано методу `whenEmpty`, которое будет выполняться, если коллекция не пуста:

```
$collection = collect(['Michael', 'Tom']);  
  
$collection->whenEmpty(function (Collection $collection) {  
    return $collection->push('Adam');  
}, function (Collection $collection) {  
    return $collection->push('Taylor');  
});  
  
$collection->all();  
  
// ['Michael', 'Tom', 'Taylor']
```

Противоположным методу `whenEmpty` является метод `whenNotEmpty`.

## whenNotEmpty()

Метод `whenNotEmpty` выполнит указанное замыкание, если коллекция не пуста:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']

$collection = collect();

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
});

$collection->all();

// []
```

Второе замыкание может быть передано методу `whenNotEmpty`, которое будет выполняться, если коллекция пуста:

```
$collection = collect();

$collection->whenNotEmpty(function (Collection $collection) {
    return $collection->push('adam');
}, function (Collection $collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

Противоположным методу `whenNotEmpty` является метод `whenEmpty`.

## where()

Метод `where` фильтрует коллекцию по указанной паре ключ / значение:

```

$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/

```

Метод `where` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `whereStrict` для фильтрации с использованием «жесткого» сравнения.

При желании вы можете передать оператор сравнения в качестве второго параметра. Поддерживаемые операторы: '`==`', '`!=`', '`!=`', '`==`', '`=`', '`<>`', '`>`', '`<`', '`>=`' и '`<=`':

```

$collection = collect([
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ['name' => 'Sue', 'deleted_at' => null],
]);

$filtered = $collection->where('deleted_at', '!=', null);

$filtered->all();

/*
[
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
]
*/

```

## whereStrict()

Этот метод имеет ту же сигнатуру, что и метод `where`; однако, все значения сравниваются с использованием «жесткого» сравнения.

## whereBetween()

Метод `whereBetween` фильтрует коллекцию, определяя, находится ли переданное значение элемента в указанном диапазоне:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
[
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]
*/
```

## whereIn()

Метод `whereIn` удаляет элементы из коллекции, у которых значения отсутствуют в указанном массиве:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);
```

```
$filtered->all();  
  
/*  
 [  
     ['product' => 'Desk', 'price' => 200],  
     ['product' => 'Bookcase', 'price' => 150],  
 ]  
*/
```

Метод `whereIn` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `whereInStrict` для фильтрации с использованием «жесткого» сравнения.

## whereInStrict()

Этот метод имеет ту же сигнатуру, что и метод `whereIn`; однако, все значения сравниваются с использованием «жесткого» сравнения.

## whereInstanceOf()

Метод `whereInstanceOf` фильтрует коллекцию по указанному типу класса:

```
use App\Models\User;  
use App\Models\Post;  
  
$collection = collect([  
    new User,  
    new User,  
    new Post,  
]);  
  
$filtered = $collection->whereInstanceOf(User::class);  
  
$filtered->all();  
  
// [App\Models\User, App\Models\User]
```

## whereNotBetween()

Метод `whereNotBetween` фильтрует коллекцию, определяя, находится ли переданное значение элемента вне указанного диапазона:

```

$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Pencil', 'price' => 30],
]
*/

```

## whereNotIn()

Метод `whereNotIn` удаляет элементы из коллекции, у которых значения присутствуют в указанном массиве:

```

$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/

```

Метод `whereNotIn` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной

целому числу того же значения. Используйте метод [whereNotInStrict](#) для фильтрации с использованием «жесткого» сравнения.

## whereNotInStrict()

Этот метод имеет ту же сигнатуру, что и метод [whereNotIn](#); однако, все значения сравниваются с использованием «жесткого» сравнения.

## whereNotNull()

Метод [whereNotNull](#) возвращает элементы из коллекции, для которых значение указанного ключа не равно `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNotNull('name');

$filtered->all();

/*
[
    ['name' => 'Desk'],
    ['name' => 'Bookcase'],
]
*/
```

## whereNull()

Метод [whereNull](#) возвращает элементы из коллекции, для которых значение указанного ключа равно `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNull('name');
```

```
$filtered->all();  
  
/*  
 [  
     ['name' => null],  
 ]  
 */
```

## wrap()

Статический метод `wrap` обворачивает указанное значение в коллекцию, если это применимо:

```
use Illuminate\Support\Collection;  
  
$collection = Collection::wrap('John Doe');  
  
$collection->all();  
  
// ['John Doe']  
  
$collection = Collection::wrap(['John Doe']);  
  
$collection->all();  
  
// ['John Doe']  
  
$collection = Collection::wrap(collect('John Doe'));  
  
$collection->all();  
  
// ['John Doe']
```

## zip()

Метод `zip` объединяет значения переданного массива со значениями исходной коллекции по их соответствующему индексу:

```
$collection = collect(['Chair', 'Desk']);  
  
$zipped = $collection->zip([100, 200]);  
  
$zipped->all();
```

```
// [['Chair', 100], ['Desk', 200]]
```

## # Сообщения высшего порядка

Коллекции также обеспечивают поддержку «сообщений высшего порядка», которые являются сокращениями для выполнения общих действий с коллекциями. Методы коллекции, которые предоставляют сообщения высшего порядка: [average](#), [avg](#), [contains](#), [each](#), [every](#), [filter](#), [first](#), [flatMap](#), [groupBy](#), [keyBy](#), [map](#), [max](#), [min](#), [partition](#), [reject](#), [skipUntil](#), [skipWhile](#), [some](#), [sortBy](#), [sortByDesc](#), [sum](#), [takeUntil](#), [takeWhile](#), и [unique](#).

К каждому сообщению высшего порядка можно получить доступ как к динамическому свойству экземпляра коллекции. Например, давайте использовать сообщение высшего порядка [each](#), вызывая метод для каждого объекта коллекции:

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Точно так же мы можем использовать сообщение высшего порядка [sum](#), чтобы собрать общее количество «голосов» для коллекции пользователей:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

## # Отложенные коллекции

### Введение в отложенные коллекции

Прежде чем узнать больше об отложенных коллекциях Laravel, потратьте некоторое время на

то, чтобы ознакомиться с [генераторами PHP](#).

В дополнении к мощному классу `Collection`, класс `LazyCollection` использует [генераторы](#) PHP, чтобы вы могли работать с очень большим набором данных при низком потреблении памяти.

Например, представьте, что ваше приложение должно обрабатывать файл журнала размером в несколько гигабайт, используя при этом методы коллекций Laravel для анализа журналов. Вместо одновременного чтения всего файла в память можно использовать отложенные коллекции, чтобы сохранить в памяти только небольшую часть файла в текущий момент:

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function (array $lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Или представьте, что вам нужно перебрать 10 000 моделей Eloquent. При использовании традиционных коллекций Laravel все 10 000 моделей Eloquent должны быть загружены в память одновременно:

```
use App\Models\User;

$users = User::all()->filter(function (User $user) {
    return $user->id > 500;
});
```

Однако, метод `cursor` построителя запросов возвращает экземпляр `LazyCollection`. Это позволяет вам по-прежнему выполнять только один запрос к базе данных, но при этом одновременно загружать в память только одну модель Eloquent. В этом

примере замыкание метода `filter` не выполнится до тех пор, пока мы на самом деле не переберем каждого пользователя индивидуально, что позволяет значительно сократить использование памяти:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

## Создание отложенных коллекций

Чтобы создать экземпляр отложенной коллекции, вы должны передать функцию генератора PHP методу `make` коллекции:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
});
```

## Контракт `Enumerable`

Почти все методы, доступные в классе `Collection`, также доступны в классе `LazyCollection`. Оба класса реализуют контракт `Illuminate\Support\Enumerable`, который определяет следующие методы:

[all\(\)](#)  
[average\(\)](#)  
[avg\(\)](#)  
[chunk\(\)](#)  
[chunkWhile\(\)](#)

collapse()  
combine()  
collect()  
concat()  
contains()  
containsStrict()  
count()  
countBy()  
crossJoin()  
dd()  
diff()  
diffAssoc()  
diffKeys()  
dump()  
duplicates()  
duplicatesStrict()  
each()  
eachSpread()  
every()  
except()  
filter()  
first()  
firstOrFail()  
firstWhere()  
flatMap()  
flatten()  
flip()  
forget()  
forPage()  
get()  
groupBy()  
has()  
implode()  
intersect()

intersectAssoc()

intersectByKeys()

isEmpty()

isNotEmpty()

join()

keyBy()

keys()

last()

macro()

make()

map()

mapInto()

mapSpread()

mapToGroups()

mapWithKeys()

max()

median()

merge()

mergeRecursive()

min()

mode()

nth()

only()

pad()

partition()

pipe()

pipeInto()

pluck()

pop()

prepend()

pull()

push()

put()

random()

reduce()  
reject()  
replace()  
replaceRecursive()  
reverse()  
search()  
shift()  
shuffle()  
skip()  
skipUntil()  
skipWhile()  
slice()  
sole()  
some()  
sort()  
sortBy()  
sortByDesc()  
sortDesc()  
sortKeys()  
sortKeysDesc()  
splice()  
split()  
splitIn()  
sum()  
take()  
takeUntil()  
takeWhile()  
tap()  
times()  
toArray()  
toJson()  
transform()  
union()  
unique()

`uniqueStrict()`  
`unless()`  
`unlessEmpty()`  
`unlessNotEmpty()`  
`unwrap()`  
`values()`  
`when()`  
`whenEmpty()`  
`whenNotEmpty()`  
`where()`  
`whereStrict()`  
`whereBetween()`  
`whereIn()`  
`whereInStrict()`  
`whereInstanceOf()`  
`whereNotBetween()`  
`whereNotIn()`  
`whereNotInStrict()`  
`whereNotNull()`  
`whereNull()`  
`wrap()`  
`zip()`

Методы, которые изменяют коллекцию (такие, как `shift`, `pop`, `prepend` и т.д.), **недоступны** в классе `LazyCollection`.

## Методы отложенных коллекций

В дополнение к методам, определенным в контракте `Enumerable`, класс `LazyCollection` содержит следующие методы:

## takeUntilTimeout()

Метод `takeUntilTimeout` возвращает новую отложенную коллекцию, которая будет перечислять значения до указанного времени. По истечении этого времени коллекция перестанет перечислять:

```
$lazyCollection = LazyCollection::times(INF)
->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function (int $number) {
    dump($number);

    sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

Чтобы проиллюстрировать использование этого метода, представьте приложение, которое отправляет счета из базы данных с помощью курсора. Вы можете определить [запланированную задачу](#), которая запускается каждые 15 минут и обрабатывает счета максимум 14 минут:

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
->takeUntilTimeout(
    Carbon::createFromTimestamp(Laravel_START)->add(14, 'minutes')
)
->each(fn (Invoice $invoice) => $invoice->submit());
```

## tapEach()

В то время как метод `each` вызывает переданное замыкание для каждого элемента в коллекции сразу же, метод `tapEach` вызывает переданное замыкание только тогда, когда элементы извлекаются из списка один за другим:

```
// Пока ничего не выведено ...
$lazyCollection = LazyCollection::times(INF)->tapEach(function (int $value) {
    dump($value);
});

// Три элемента выведено ...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

## throttle()

Метод `throttle` будет регулировать ленивую коллекцию таким образом, чтобы каждое значение возвращалось через указанное количество секунд. Этот метод особенно полезен в ситуациях, когда вы можете взаимодействовать с внешними API, которые ограничивают скорость входящих запросов:

```
use App\Models\User;

User::where('vip', true)
    ->cursor()
    ->throttle(seconds: 1)
    ->each(function (User $user) {
        // Call external API...
    });
});
```

## remember()

Метод `remember` возвращает новую отложенную коллекцию, запоминающую любые значения, которые уже были перечислены, и не будет извлекать их снова при последующих перечислениях коллекции:

```
// Запрос еще не выполнен ...
$users = User::cursor()->remember();

// Запрос выполнен ...
// Первые 5 пользователей из базы данных включены в результирующий набор ...
$users->take(5)->all();

// Первые 5 пользователей пришли из кеша коллекции ...
```

```
// Остальные из базы данных включены в результирующий набор ...
$users->take(20)->all();
```

# Параллелизм

- # Введение
- # Запуск параллельных задач
- # Отсрочка параллельных задач

## # Введение

Фасад Laravel [Concurrency](#) в настоящее время находится в стадии бета-тестирования, пока мы собираем отзывы сообщества.

Иногда вам может потребоваться выполнить несколько медленных задач, не зависящих друг от друга. Во многих случаях существенного повышения производительности можно добиться, выполняя задачи одновременно. Фасад Laravel [Concurrency](#) предоставляет простой и удобный API для одновременного выполнения замыканий.

## Совместимость параллелизма

Если вы обновились до Laravel 11.x из приложения Laravel 10.x, вам может потребоваться добавить [ConcurrencyServiceProvider](#) в массив `providers` в файле конфигурации `config/app.php` вашего приложения:

```
'providers' => ServiceProvider::defaultProviders()->merge([
    /*
     * Package Service Providers...
     */
    Illuminate\Concurrency\ConcurrencyServiceProvider::class, // [tl! add]

    /*
     * Application Service Providers...
     */
])
```

```
App\Providers\AppServiceProvider::class,  
App\Providers\AuthServiceProvider::class,  
// App\Providers\BroadcastServiceProvider::class,  
App\Providers\EventServiceProvider::class,  
App\Providers\RouteServiceProvider::class,  
])->toArray(),
```

## Как это работает

Laravel обеспечивает параллелизм путем сериализации заданных замыканий и отправки их скрытой команде Artisan CLI, которая десериализует замыкания и вызывает их в собственном PHP-процессе. После вызова замыкания полученное значение сериализуется обратно в родительский процесс.

Фасад [Concurrency](#) поддерживает три драйвера: `process` (по умолчанию), `fork` и `sync`.

Драйвер `fork` обеспечивает улучшенную производительность по сравнению с драйвером по умолчанию `process`, но его можно использовать только в контексте CLI PHP, поскольку PHP не поддерживает разветвление во время веб-запросов. Перед использованием драйвера `fork` вам необходимо установить пакет [spatie/fork](#):

```
composer require spatie/fork
```

Драйвер `sync` в первую очередь полезен во время тестирования, когда вы хотите отключить весь параллелизм и просто последовательно выполнить заданные замыкания внутри родительского процесса.

## # Запуск параллельных задач

Для запуска параллельных задач вы можете вызвать метод `run` фасада [Concurrency](#). Метод `run` принимает массив замыканий, которые должны выполняться одновременно в дочерних процессах PHP:

```
use Illuminate\Support\Facades\Concurrency;  
use Illuminate\Support\Facades\DB;  
  
[$userCount, $orderCount] = Concurrency::run([  
    fn () => DB::table('users')->count(),
```

```
fn () => DB::table('orders')->count(),
]);
```

Чтобы использовать конкретный драйвер, вы можете использовать метод `driver`:

```
$results = Concurrency::driver('fork')->run(...);
```

Или, чтобы изменить драйвер параллелизма по умолчанию, вам следует опубликовать файл конфигурации `concurrency` с помощью Artisan-команды `config:publish` и обновить параметр `default` в файле:

```
php artisan config:publish concurrency
```

## # Отсрочка параллельных задач

Если вы хотите одновременно выполнить массив замыканий, но вас не интересуют результаты, возвращаемые этими замыканиями, вам следует рассмотреть возможность использования метода `defer`. Когда вызывается метод `defer`, данные замыкания не выполняются немедленно. Вместо этого Laravel будет выполнять замыкания одновременно после отправки пользователю HTTP-ответа:

```
use App\Services\Metrics;
use Illuminate\Support\Facades\Concurrency;

Concurrency::defer([
    fn () => Metrics::report('users'),
    fn () => Metrics::report('orders'),
]);
```

## # Введение

# Как это работает

## # Захват контекста

# Стеки

## # Получение контекста

# Определение существования элемента

## # Удаление контекста

## # Скрытый контекст

## # События

# Обезвоживание

# Гидратация

# # Введение

«Контекстные» возможности Laravel позволяют вам собирать, извлекать и обмениваться информацией в запросах, заданиях и командах, выполняющихся в вашем приложении. Эта собранная информация также включается в журналы, записываемые вашим приложением, что дает вам более глубокое представление об истории выполнения окружающего кода, произошедшей до того, как была записана запись в журнале, и позволяет отслеживать потоки выполнения во всей распределенной системе.

## Как это работает

Лучший способ понять контекстные возможности Laravel — увидеть его в действии, используя встроенные функции ведения журнала. Для начала вы можете [добавить информацию в контекст](#), используя фасад `Context`. В этом примере мы будем использовать [посредника](#) для добавления URL-адреса запроса и уникального идентификатора трассировки в контекст каждого входящего запроса:

```
<?php
```

```
namespace App\Http\Middleware;
```

```
use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Context;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AddContext
{
    /**
     * Handle an incoming request.
     */
    public function handle(Request $request, Closure $next): Response
    {
        Context::add('url', $request->url());
        Context::add('trace_id', Str::uuid()->toString());

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

Информация, добавленная в контекст, автоматически добавляется в виде метаданных ко всем [записям журнала](#), которые записываются на протяжении всего запроса. Добавление контекста в виде метаданных позволяет отличать информацию, передаваемую в отдельные записи журнала, от информации, передаваемой через [Context](#). Например, представьте, что мы пишем следующую запись в журнале:

```
Log::info('User authenticated.', ['auth_id' => Auth::id()]);
```

Запись в журнале будет содержать переданный [auth\\_id](#), но запись также будет содержать [url](#) и [trace\\_id](#) контекста в качестве метаданных:

```
User authenticated. {"auth_id":27} {"url":"https://example.com/login","trace_id":"e0c...
```

Информация, добавленная в контекст, также становится доступной для заданий, отправленных в очередь. Например, представьте, что мы отправляем задание [ProcessPodcast](#) в очередь после добавления некоторой информации в контекст:

```
// In our middleware...
Context::add('url', $request->url());
Context::add('trace_id', Str::uuid()->toString());

// In our controller...
ProcessPodcast::dispatch($podcast);
```

При отправке задания любая информация, хранящаяся в данный момент в контексте, фиксируется и передается заданию. Собранная информация затем возвращается в текущий контекст во время выполнения задания. Итак, если бы метод `handle` нашего задания заключался в записи в журнал:

```
class ProcessPodcast implements ShouldQueue
{
    use Queueable;

    // ...

    /**
     * Execute the job.
     */
    public function handle(): void
    {
        Log::info('Processing podcast.', [
            'podcast_id' => $this->podcast->id,
        ]);

        // ...
    }
}
```

Результирующая запись журнала будет содержать информацию, которая была добавлена в контекст во время запроса, который первоначально отправил задание:

```
Processing podcast. {"podcast_id":95} {"url":"https://example.com/login","trace_id":'
```



Хотя мы сосредоточились на встроенных функциях контекста Laravel, связанных с ведением журнала, следующая документация покажет, как контекст позволяет вам обмениваться информацией через границу HTTP-запроса/задания в очереди и

даже как добавлять [данные скрытого контекста](#hidden- контекст), который не записывается в записи журнала.

## # Захват контекста

Вы можете хранить информацию в текущем контексте, используя метод `add` фасада `Context`:

```
use Illuminate\Support\Facades\Context;

Context::add('key', 'value');
```

Чтобы добавить несколько элементов одновременно, вы можете передать ассоциативный массив методу `add`:

```
Context::add([
    'first_key' => 'value',
    'second_key' => 'value',
]);
```

Метод `add` переопределит любое существующее значение, имеющее тот же ключ. Если вы хотите добавить информацию в контекст только в том случае, если ключ еще не существует, вы можете использовать метод `addIf`:

```
Context::add('key', 'first');

Context::get('key');
// "first"

Context::addIf('key', 'second');

Context::get('key');
// "first"
```

## Условный контекст

Метод `when` можно использовать для добавления данных в контекст на основе заданного условия. Первое замыкание, предоставленное методу `when`, будет

вызвано, если данное условие оценивается как `true`, а второе замыкание будет вызвано, если условие оценивается как `false`:

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Context;

Context::when(
    Auth::user()->isAdmin(),
    fn ($context) => $context->add('permissions', Auth::user()->permissions),
    fn ($context) => $context->add('permissions', []),
);
);
```

## Стеки

Контекст предлагает возможность создавать «стеки», которые представляют собой списки данных, хранящихся в том порядке, в котором они были добавлены. Вы можете добавить информацию в стек, вызвав метод `push`:

```
use Illuminate\Support\Facades\Context;

Context::push('breadcrumbs', 'first_value');

Context::push('breadcrumbs', 'second_value', 'third_value');

Context::get('breadcrumbs');
// [
//     'first_value',
//     'second_value',
//     'third_value',
// ]
```

Стеки могут быть полезны для сбора исторической информации о запросе, например событий, происходящих в вашем приложении. Например, вы можете создать прослушиватель событий, который будет помещать в стек каждый раз при выполнении запроса, фиксируя SQL-запрос и его продолжительность в виде кортежа:

```
use Illuminate\Support\Facades\Context;
use Illuminate\Support\Facades\DB;

DB::listen(function ($event) {
```

```
Context::push('queries', [$event->time, $event->sql]);  
});
```

Вы можете определить, находится ли значение в стеке, используя методы `stackContains` и `hiddenStackContains`:

```
if (Context::stackContains('breadcrumbs', 'first_value')) {  
    //  
}  
  
if (Context::hiddenStackContains('secrets', 'first_value')) {  
    //  
}
```

Методы `stackContains` и `hiddenStackContains` также принимают замыкание в качестве второго аргумента, что позволяет лучше контролировать операцию сравнения значений:

```
use Illuminate\Support\Facades\Context;  
use Illuminate\Support\Str;  
  
return Context::stackContains('breadcrumbs', function ($value) {  
    return Str::startsWith($value, 'query_');  
});
```

## # Получение контекста

Вы можете получить информацию из контекста, используя метод `get` фасада `Context`:

```
use Illuminate\Support\Facades\Context;  
  
$value = Context::get('key');
```

Метод `only` можно использовать для получения подмножества информации в контексте:

```
$data = Context::only(['first_key', 'second_key']);
```

Метод `pull` можно использовать для извлечения информации из контекста и немедленного удаления ее из контекста:

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

Если вы хотите получить всю информацию, хранящуюся в контексте, вы можете вызвать метод `all`:

```
$data = Context::all();
```

## Определение существования элемента

Вы можете использовать метод `has`, чтобы определить, имеет ли контекст какое-либо значение, сохраненное для данного ключа:

```
use Illuminate\Support\Facades\Context;

if (Context::has('key')) {
    // ...
}
```

Метод `has` вернет `true` независимо от сохраненного значения. Так, например, ключ со значением `null` будет считаться присутствующим:

```
Context::add('key', null);

Context::has('key');
// true
```

## # Удаление контекста

Метод `forget` можно использовать для удаления ключа и его значения из текущего контекста:

```
use Illuminate\Support\Facades\Context;

Context::add(['first_key' => 1, 'second_key' => 2]);

Context::forget('first_key');

Context::all();

// ['second_key' => 2]
```

Вы можете забыть несколько ключей одновременно, предоставив массив методу `forget`:

```
Context::forget(['first_key', 'second_key']);
```

## # Скрытый контекст

Контекст предлагает возможность хранить «скрытые» данные. Эта скрытая информация не добавляется в журналы и недоступна с помощью описанных выше методов получения данных. Контекст предоставляет другой набор методов для взаимодействия со скрытой контекстной информацией:

```
use Illuminate\Support\Facades\Context;

Context::addHidden('key', 'value');

Context::getHidden('key');
// 'value'

Context::get('key');
// null
```

«Скрытые» методы отражают функциональность нескрытых методов, описанных выше:

```
Context::addHidden(/* ... */);
Context::addHiddenIf(/* ... */);
Context::pushHidden(/* ... */);
Context::getHidden(/* ... */);
Context::pullHidden(/* ... */);
```

```
Context::onlyHidden(/* ... */);
Context::allHidden(/* ... */);
Context::hasHidden(/* ... */);
Context::forgetHidden(/* ... */);
```

## # События

Контекст отправляет два события, которые позволяют вам подключиться к процессу гидратации и обезвоживания контекста.

Чтобы проиллюстрировать, как можно использовать эти события, представьте, что в промежуточном программном обеспечении вашего приложения вы устанавливаете значение конфигурации `app.locale` на основе заголовка `Accept-Language` входящего HTTP-запроса. События контекста позволяют вам захватить это значение во время запроса и восстановить его в очереди, гарантируя, что отправляемые в очередь уведомления имеют правильное значение `app.locale`. Для достижения этой цели мы можем использовать события контекста и данные `hidden`, что будет показано в следующей документации.

## Обезвоживание

Всякий раз, когда задание отправляется в очередь, данные в контексте «обезвоживаются» и фиксируются вместе с полезной нагрузкой задания. Метод `Context::dehydrating` позволяет вам зарегистрировать замыкание, которое будет вызываться во время процесса обезвоживания. В рамках этого закрытия вы можете вносить изменения в данные, которые будут доступны для задания в очереди.

Обычно вам следует зарегистрировать `dehydrating` обратные вызовы в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Log\Context\Repository;
use Illuminate\Support\Facades\Config;
use Illuminate\Support\Facades\Context;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Context::dehydrating(function (Repository $context) {
```

```
$context->addHidden('locale', Config::get('app.locale'));  
});  
}
```

Не следует использовать фасад `Context` в обратном вызове `dehydrating`, так как это изменит контекст текущего процесса. Убедитесь, что вы вносите изменения только в репозиторий, переданный в обратный вызов.

## Гидратация

Всякий раз, когда поставленное в очередь задание начинает выполняться в очереди, любой контекст, который был общим с заданием, будет «гидратирован» обратно в текущий контекст. Метод `Context::hydrated` позволяет вам зарегистрировать замыкание, которое будет вызываться во время процесса гидратации.

Обычно вам следует регистрировать `hydrated` обратные вызовы в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Log\Context\Repository;  
use Illuminate\Support\Facades\Config;  
use Illuminate\Support\Facades\Context;  
  
/**  
 * Bootstrap any application services.  
 */  
public function boot(): void  
{  
    Context::hydrated(function (Repository $context) {  
        if ($context->hasHidden('locale')) {  
            Config::set('app.locale', $context->getHidden('locale'));  
        }  
    });  
}
```

Не следует использовать фасад `Context` в обратном вызове `hydrated` и вместо этого убедитесь, что вы вносите изменения только в репозиторий, переданный в обратный вызов.

# Контракты

## # Введение

# Контракты против Фасадов

## # Когда использовать контракты

## # Как использовать контракты

## # Справочник контрактов

## # Введение

«Контракты» Laravel – это набор интерфейсов, которые определяют основные службы фреймворка. Например, контракт `Illuminate\Contracts\Queue\Queue` определяет методы, необходимые для отправки заданий в очередь, а контракт `Illuminate\Contracts\Mail\Mailer` – для отправки электронной почты.

Каждый контракт имеет соответствующую реализацию, предоставляемую фреймворком. Например, Laravel предлагает реализацию очереди с множеством драйверов и реализацию компонента для отправки почты, который работает на базе [Symfony Mailer](#).

Все контракты Laravel хранятся в [собственном репозитории](#) GitHub. Это обеспечивает быстрый доступ к списку всех доступных контрактов, а также единый, отдельный пакет, который используется разработчиками пакетов, взаимодействующих со службами Laravel.

## Контракты против Фасадов

[Фасады](#) и глобальные хелперы Laravel обеспечивают простой способ использования сервисов Laravel без необходимости объявления типа зависимости(Type Hinting) и извлечения контракта из сервис-контейнера. В большинстве случаев каждый фасад имеет эквивалентный контракт.

В отличие от фасадов, которые не требуют инициализации в конструкторе вашего класса, контракты позволяют вам определять явные зависимости для ваших

классов. Разработчики, которые предпочитают явно определять зависимости, используют контракты, некоторые разработчики пользуются удобством фасадов. **В целом, при разработке большинства приложений можно без проблем использовать фасады.**

## # Когда использовать контракты

Решение об использовании контрактов или фасадов будет зависеть от личного вкуса и вкусов вашей команды. И контракты, и фасады могут использоваться для создания надежных, хорошо тестируемых приложений Laravel. Контракты и фасады не исключают друг друга. Некоторые части ваших приложений могут использовать фасады, а другие зависеть от контрактов. Пока вы сосредоточены на реализации обязанностей класса, вы не заметите практических различий между использованием контрактов и фасадов.

При разработке большинства приложений можно без проблем использовать фасады. Если вы создаете пакет, который будет интегрирован с несколькими PHP-фреймворками, вы можете указать пакет [illuminate/contracts](#) в файле `composer.json` вашего пакета для определения вашей интеграции со службами Laravel без необходимости требовать конкретную реализацию для Laravel.

## # Как использовать контракты

Как получить реализацию контракта? На самом деле это довольно просто.

Многие типы классов в Laravel извлекаются из [сервис-контейнера](#), включая контроллеры, слушатели событий, посредники, очереди заданий и даже замыкания маршрутов. Итак, чтобы получить реализацию контракта, вы можете просто внедрить интерфейс в конструктор извлекаемого класса.

Например, взгляните на этот слушатель:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderWasPlaced;  
use App\Models\User;  
use Illuminate\Contracts\Redis\Factory;
```

```

class CacheOrderInformation
{
    /**
     * Создать новый экземпляр обработчика события.
     */
    public function __construct(
        protected Factory $redis,
    ) {}

    /**
     * Обработать событие.
     */
    public function handle(OrderWasPlaced $event) : void
    {
        // ...
    }
}

```

Когда слушатель события будет извлечен, сервис-контейнер, используя инициализацию типов в конструкторе класса, внедрит соответствующую зависимость. Чтобы узнать больше о регистрации в сервис-контейнере, ознакомьтесь с [его документацией](#).

## # Справочник контрактов

В этой таблице содержится краткий справочник контрактов и эквивалентных им фасадов Laravel:

Контракт	Фасад
<a href="#">Illuminate\Contracts\Auth\Access\Authorizable</a>	
<a href="#">Illuminate\Contracts\Auth\Access\Gate</a>	Gate
<a href="#">Illuminate\Contracts\Auth\Authenticatable</a>	
<a href="#">Illuminate\Contracts\Auth\CanResetPassword</a>	
<a href="#">Illuminate\Contracts\Auth\Factory</a>	Auth
<a href="#">Illuminate\Contracts\Auth\Guard</a>	Auth::guard()

<b>Контракт</b>	<b>Фасад</b>
<a href="#">Illuminate\Contracts\Auth\PasswordBroker</a>	Password::broker()
<a href="#">Illuminate\Contracts\Auth\PasswordBrokerFactory</a>	Password
<a href="#">Illuminate\Contracts\Auth\StatefulGuard</a>	
<a href="#">Illuminate\Contracts\Auth\SupportsBasicAuth</a>	
<a href="#">Illuminate\Contracts\Auth\UserProvider</a>	
<a href="#">Illuminate\Contracts\Broadcasting\Broadcaster</a>	Broadcast::connection()
<a href="#">Illuminate\Contracts\Broadcasting\Factory</a>	Broadcast
<a href="#">Illuminate\Contracts\Broadcasting\ShouldBroadcast</a>	
<a href="#">Illuminate\Contracts\Broadcasting\ShouldBroadcastNow</a>	
<a href="#">Illuminate\Contracts\Bus\Dispatcher</a>	Bus
<a href="#">Illuminate\Contracts\Bus\QueueingDispatcher</a>	Bus::dispatchToQueue()
<a href="#">Illuminate\Contracts\Cache\Factory</a>	Cache
<a href="#">Illuminate\Contracts\Cache\Lock</a>	
<a href="#">Illuminate\Contracts\Cache\LockProvider</a>	
<a href="#">Illuminate\Contracts\Cache\Repository</a>	Cache::driver()
<a href="#">Illuminate\Contracts\Cache\Store</a>	
<a href="#">Illuminate\Contracts\Config\Repository</a>	Config
<a href="#">Illuminate\Contracts\Console\Application</a>	
<a href="#">Illuminate\Contracts\Console\Kernel</a>	Artisan
<a href="#">Illuminate\Contracts\Container\Container</a>	App

<b>Контракт</b>	<b>Фасад</b>
<a href="#">Illuminate\Contracts\Cookie\Factory</a>	Cookie
<a href="#">Illuminate\Contracts\Cookie\QueueingFactory</a>	Cookie::queue()
<a href="#">Illuminate\Contracts\Database\ModelIdentifier</a>	
<a href="#">Illuminate\Contracts\Debug\ExceptionHandler</a>	
<a href="#">Illuminate\Contracts\Encryption\Encrypter</a>	Crypt
<a href="#">Illuminate\Contracts\Events\Dispatcher</a>	Event
<a href="#">Illuminate\Contracts\Filesystem\Cloud</a>	Storage::cloud()
<a href="#">Illuminate\Contracts\Filesystem\Factory</a>	Storage
<a href="#">Illuminate\Contracts\Filesystem\Filesystem</a>	Storage::disk()
<a href="#">Illuminate\Contracts\Foundation\Application</a>	App
<a href="#">Illuminate\Contracts\Hashing\Hasher</a>	Hash
<a href="#">Illuminate\Contracts\Http\Kernel</a>	
<a href="#">Illuminate\Contracts\Mail\Mailable</a>	
<a href="#">Illuminate\Contracts\Mail\Mailer</a>	Mail
<a href="#">Illuminate\Contracts\Mail\MailQueue</a>	Mail::queue()
<a href="#">Illuminate\Contracts\Notifications\Dispatcher</a>	Notification
<a href="#">Illuminate\Contracts\Notifications\Factory</a>	Notification
<a href="#">Illuminate\Contracts\Pagination\LengthAwarePaginator</a>	
<a href="#">Illuminate\Contracts\Pagination\Paginator</a>	
<a href="#">Illuminate\Contracts\Pipeline\Hub</a>	

<b>Контракт</b>	<b>Фасад</b>
<a href="#">Illuminate\Contracts\Pipeline\Pipeline</a>	Pipeline;
<a href="#">Illuminate\Contracts\Queue\EntityResolver</a>	
<a href="#">Illuminate\Contracts\Queue\Factory</a>	Queue
<a href="#">Illuminate\Contracts\Queue\Job</a>	
<a href="#">Illuminate\Contracts\Queue\Monitor</a>	Queue
<a href="#">Illuminate\Contracts\Queue\Queue</a>	Queue::connection()
<a href="#">Illuminate\Contracts\Queue\QueueableCollection</a>	
<a href="#">Illuminate\Contracts\Queue\QueueableEntity</a>	
<a href="#">Illuminate\Contracts\Queue\ShouldQueue</a>	
<a href="#">Illuminate\Contracts\Redis\Factory</a>	Redis
<a href="#">Illuminate\Contracts\Routing\BindingRegistrar</a>	Route
<a href="#">Illuminate\Contracts\Routing\Registrar</a>	Route
<a href="#">Illuminate\Contracts\Routing\ResponseFactory</a>	Response
<a href="#">Illuminate\Contracts\Routing\UrlGenerator</a>	URL
<a href="#">Illuminate\Contracts\Routing\UrlRoutable</a>	
<a href="#">Illuminate\Contracts\Session\Session</a>	Session::driver()
<a href="#">Illuminate\Contracts\Support\Arrayable</a>	
<a href="#">Illuminate\Contracts\Support\Htmlable</a>	
<a href="#">Illuminate\Contracts\Support\Jsonable</a>	
<a href="#">Illuminate\Contracts\Support\MessageBag</a>	

<b>Контракт</b>	<b>Фасад</b>
<a href="#">Illuminate\Contracts\Support\MessageProvider</a>	
<a href="#">Illuminate\Contracts\Support\Renderable</a>	
<a href="#">Illuminate\Contracts\Support\Responsable</a>	
<a href="#">Illuminate\Contracts\Translation\Loader</a>	
<a href="#">Illuminate\Contracts\Translation\Translator</a>	Lang
<a href="#">Illuminate\Contracts\Validation\Factory</a>	Validator
<a href="#">Illuminate\Contracts\Validation\ValidatesWhenResolved</a>	
<a href="#">Illuminate\Contracts\Validation\ValidationRule</a>	
<a href="#">Illuminate\Contracts\Validation\Validator</a>	Validator::make()
<a href="#">Illuminate\Contracts\View\Engine</a>	
<a href="#">Illuminate\Contracts\View\Factory</a>	View
<a href="#">Illuminate\Contracts\View\View</a>	View::make()

# События (Events)

## # Введение

## # Генерация событий и слушателей

## # Регистрация событий и слушателей

# Автопоиск событий

# Ручная регистрация событий

# Слушатели на основе замыкания

## # Определение событий

## # Определение слушателей

## # Слушатели событий в очереди

# Взаимодействие с очередью вручную

# Слушатели событий в очереди и транзакции базы данных

# Обработка невыполненных заданий

## # Отправка событий

# Отправка событий после транзакций в базе данных

## # Подписчики событий

# Написание подписчиков на события

# Регистрация подписчиков на события

## # Тестирование

# Подмена определенного набора событий

# Подмена событий в ограниченной области видимости

## # Введение

События Laravel обеспечивают простую реализацию шаблона Наблюдатель, позволяя вам подписываться и отслеживать различные события, происходящие в вашем приложении. Классы событий обычно хранятся в каталоге `app/Events`, а их слушатели – в `app/Listeners`. Не беспокойтесь, если вы не видите эти каталоги в своем приложении, так как они будут созданы для вас, когда вы будете генерировать события и слушатели с помощью команд консоли Artisan.

События служат отличным способом разделения различных аспектов вашего приложения, поскольку одно событие может иметь несколько слушателей, которые не зависят друг от друга. Например, бывает необходимо отправлять уведомление Slack своему пользователю каждый раз, когда заказ будет отправлен. Вместо того чтобы связывать код обработки заказа с кодом уведомления Slack, вы можете вызвать событие `App\\Events\OrderShipped`, которое слушатель может получить и использовать для отправки уведомления Slack.

## # Генерация событий и слушателей

Чтобы быстро генерировать события и слушателей, вы можете использовать Artisan-команды `make:event` и `make:listener`:

```
php artisan make:event PodcastProcessed
```

```
php artisan make:listener SendPodcastNotification --event=PodcastProcessed
```

For convenience, you may also invoke the `make:event` and `make:listener` Artisan commands without additional arguments. When you do so, Laravel will automatically prompt you for the class name and, when creating a listener, the event it should listen to: Для удобства вы также можете вызывать команды Artisan `make:event` и `make:listener` без дополнительных аргументов. Когда вы это сделаете, Laravel автоматически предложит вам ввести имя класса и, при создании слушателя, событие, которое он должен прослушивать:

```
php artisan make:event
```

```
php artisan make:listener
```

## # Регистрация событий и слушателей

### Автопоиск событий

По умолчанию Laravel автоматически найдет и зарегистрирует ваших слушателей событий, просканировав каталог `Listeners` вашего приложения. Когда Laravel находит какой-либо метод класса слушателя, который начинается с `handle` или

\_\_invoke, Laravel регистрирует эти методы как слушатели событий для события, тип которого указан в сигнатуре метода:

```
use App\Events\PodcastProcessed;

class SendPodcastNotification
{
    /**
     * Handle the given event.
     */
    public function handle(PodcastProcessed $event): void
    {
        // ...
    }
}
```

Вы можете прослушивать несколько событий, используя типы объединения PHP:

```
/**
 * Handle the given event.
 */
public function handle(PodcastProcessed|PodcastPublished $event): void
{
    // ...
}
```

Если вы планируете хранить свои слушатели в другом каталоге или в нескольких каталогах, вы можете поручить Laravel сканировать эти каталоги с помощью метода `withEvents` в файле `bootstrap/app.php` вашего приложения:

```
->withEvents(discover: [
    __DIR__.'/../app/Domain/Orders/Listeners',
])
```

Команда `event:list` может использоваться для вывода списка всех слушателей, зарегистрированных в вашем приложении:

```
php artisan event:list
```

## Кэширование событий

Чтобы повысить скорость вашего приложения, вам следует кэшировать манифест всех прослушивателей вашего приложения с помощью Artisan-команд `optimize` или `event:cache`. Обычно эту команду следует запускать как часть [процесса развертывания](#). Этот манифест будет использоваться платформой для ускорения процесса регистрации событий. Команда `event:clear` может использоваться для удаления кэша событий.

## Ручная регистрация событий

Используя фасад `Event`, вы можете вручную регистрировать события и соответствующие им слушателей в методе `boot AppServiceProvider` вашего приложения:

```
use App\Domain\Orders\Events\PodcastProcessed;
use App\Domain\Orders\Listeners\SendPodcastNotification;
use Illuminate\Support\Facades\Event;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Event::listen(
        PodcastProcessed::class,
        SendPodcastNotification::class,
    );
}
```

Команда `event:list` может использоваться для вывода списка всех слушателей, зарегистрированных в вашем приложении:

```
php artisan event:list
```

## Слушатели на основе замыкания

Обычно слушатели определяются как классы; однако вы также можете вручную зарегистрировать слушателей событий на основе замыканий в методе `boot` вашего приложения `AppServiceProvider`:

```
use App\Events\PodcastProcessed;
use Illuminate\Support\Facades\Event;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Event::listen(function (PodcastProcessed $event) {
        // ...
    });
}
```

## Анонимные слушатели событий в очереди

При регистрации слушателей событий на основе замыкания вы можете обернуть замыкание слушателя в функцию `Illuminate\Events\queueable`, чтобы указать Laravel выполнить слушателя с использованием [очереди](#):

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        // ...
    }));
}
```

Как и в случае с заданиями в очередях, вы можете использовать методы `onConnection`, `onQueue` и `delay` для детализации выполнения слушателя в очереди:

```
Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10));
```

Если вы хотите обрабатывать сбои анонимного слушателя в очереди, то вы можете передать замыкание методу `catch` при определении слушателя `queueable`.

Это замыкание получит экземпляр события и экземпляр `Throwable`, вызвавший сбой слушателя:

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
use Throwable;

Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->catch(function (PodcastProcessed $event, Throwable $e) {
    // Событие в очереди завершилось неудачно ...
}));
```

## Анонимные слушатели группы событий

Вы также можете зарегистрировать слушателей, используя символ `*` в качестве подстановочного параметра, что позволит вам перехватывать несколько событий на одном слушателе. Слушатели, зарегистрированные с помощью данного синтаксиса, получают имя события в качестве первого аргумента и весь массив данных события в качестве второго аргумента:

```
Event::listen('event.*', function (string $eventName, array $data) {
    // ...
});
```

## # Определение событий

Класс событий – это, по сути, контейнер данных, который содержит информацию, относящуюся к событию. Например, предположим, что событие `App\Events\OrderShipped` получает объект [Eloquent ORM](#):

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;
```

```
class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Создать новый экземпляр события.
     */
    public function __construct(
        public Order $order,
    ) {}

}
```

Как видите, в этом классе событий нет логики. Это контейнер для экземпляра `App\Models\Order` заказа, который был выполнен. Трейт `SerializesModels`, используемый событием, будет изящно сериализовать любые модели Eloquent, если объект события сериализуется с использованием функции `serialize` PHP, например, при использовании [слушателей в очереди](#).

## # Определение слушателей

Затем, давайте посмотрим на слушателя для нашего примера события. Слушатели событий получают экземпляры событий в своем методе `handle`. Команда Artisan `make:listener` при вызове с опцией `--event` автоматически импортирует соответствующий класс события и указывает тип события в методе `handle`. В методе `handle` вы можете выполнять любые действия, необходимые для реагирования на событие:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * Создать слушателя событий.
     */
    public function __construct() {}

    /**
     * Обработать событие.
     */
}
```

```
public function handle(OrderShipped $event): void
{
    // Доступ к заказу с помощью `{$event->order}` ...
}
```

В конструкторе ваших слушателей событий могут быть объявлены любые необходимые типы зависимостей. Все слушатели событий разрешаются через [контейнер служб](#) Laravel, поэтому зависимости будут внедрены автоматически.

## Остановка распространения события

По желанию можно остановить распространение события среди других слушателей. Вы можете сделать это, вернув `false` из метода `handle` вашего слушателя.

## # Слушатели событий в очереди

Слушатели в очереди могут быть полезны, если ваш слушатель собирается выполнять медленную задачу, такую как отправка электронной почты или выполнение HTTP-запроса. Перед использованием слушателей в очереди убедитесь, что вы [сконфигурировали очередь](#) и запустили обработчик очереди на вашем сервере или в локальной среде разработки.

Чтобы указать, что слушатель должен быть поставлен в очередь, добавьте интерфейс `ShouldQueue` в класс слушателя. Слушатели, сгенерированные командами `event:generate` и `make:listener` Artisan, уже будут иметь этот интерфейс, импортируемый в текущее пространство имен, поэтому вы можете использовать его немедленно:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
```

```
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    // ...
}
```

Это все! Теперь, когда отправляется событие, обрабатываемое этим слушателем, слушатель автоматически ставится в очередь диспетчером событий с использованием [системы очередей](#) Laravel. Если при выполнении слушателя в очереди не возникает никаких исключений, задание в очереди будет автоматически удалено после завершения обработки.

## Настройка соединения очереди, имени, и времени задержки

Если вы хотите настроить соединение очереди, имя очереди или время задержки очереди для слушателя событий, то вы можете определить свойства `$connection`, `$queue`, или `$delay` в своем классе слушателя:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * Имя соединения, на которое должно быть отправлено задание.
     *
     * @var string|null
     */
    public $connection = 'sq';

    /**
     * Имя очереди, в которую должно быть отправлено задание.
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**

```

```

    * Время (в секундах) до обработки задания.
    *
    * @var int
    */
public $delay = 60;
}

```

Если вы хотите определить соединение очереди слушателя или имя очереди слушателя во время выполнения, вы можете определить методы `viaConnection`, `viaQueue` или `withDelay` слушателя:

```

/**
 * Получить имя подключения очереди слушателя.
 */
public function viaConnection(): string
{
    return 'sq';
}

/**
 * Получить имя очереди слушателя.
 */
public function viaQueue(): string
{
    return 'listeners';
}

/**
 * Получить количество секунд до того, как задача должна быть выполнена.
 */
public function withDelay(OrderShipped $event): int
{
    return $event->highPriority ? 0 : 60;
}

```

## Условная отправка слушателей в очередь

Иногда требуется определить, следует ли ставить слушателя в очередь на основе некоторых данных, доступных только во время выполнения. Для этого к слушателю может быть добавлен метод `shouldQueue`, чтобы определить, следует ли поставить слушателя в очередь. Если метод `shouldQueue` возвращает `false`, то слушатель не будет поставлен в очередь:

```
<?php

namespace App\Listeners;

use App\Events\OrderCreated;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * Наградить покупателя подарочной картой.
     */
    public function handle(OrderCreated $event): void
    {
        // ...
    }

    /**
     * Определить, следует ли ставить слушателя в очередь.
     */
    public function shouldQueue(OrderCreated $event): bool
    {
        return $event->order->subtotal >= 5000;
    }
}
```

## Взаимодействие с очередью вручную

Если вам нужно вручную получить доступ к методам `delete` и `release` базового задания в очереди слушателя, вы можете сделать это с помощью трейта `Illuminate\Queue\InteractsWithQueue`. Этот трейт по умолчанию импортируется в сгенерированные слушатели и обеспечивает доступ к этим методам:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;
```

```

    /**
     * Обработать событие.
     */
    public function handle(OrderShipped $event): void
    {
        if (true) {
            $this->release(30);
        }
    }
}

```

## Слушатели событий в очереди и транзакции базы данных

Когда слушатели в очереди отправляются в транзакциях базы данных, они могут быть обработаны очередью до того, как транзакция базы данных будет зафиксирована. Когда это происходит, любые обновления, внесенные вами в модели или записи базы данных во время транзакции базы данных, могут еще не быть отражены в базе данных. Кроме того, любые модели или записи базы данных, созданные в рамках транзакции, могут не существовать в базе данных. Если ваш слушатель зависит от этих моделей, могут возникнуть непредвиденные ошибки при обработке задания, которое отправляет поставленный в очередь слушатель.

Если опция `after_commit` вашего соединения с очередью установлена в значение `false`, то вы все равно можете указать, что конкретный слушатель в очереди должен быть выполнен после того, как все открытые транзакции в базе данных будут завершены, реализовав интерфейс `ShouldQueueAfterCommit` в классе слушателя:

```

<?php

namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueueAfterCommit;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueueAfterCommit
{
    use InteractsWithQueue;
}

```

Чтобы узнать больше о том, как обойти эти проблемы, просмотрите документацию, касающуюся [заданий в очереди и транзакций базы данных](#).

## Обработка невыполненных заданий

Иногда ваши слушатели событий в очереди могут дать сбой. Если слушатель в очереди превышает максимальное количество попыток, определенное вашим обработчиком очереди, для вашего слушателя будет вызван метод `failed`. Метод `failed` получает экземпляр события и `Throwable`, вызвавший сбой:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Throwable;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Обработать событие.
     */
    public function handle(OrderShipped $event): void
    {
        // ...
    }

    /**
     * Обработать провал задания.
     */
    public function failed(OrderShipped $event, Throwable $exception): void
    {
        // ...
    }
}
```

# Указание максимального количества

## попыток слушателя в очереди

Если один из ваших слушателей в очереди обнаруживает ошибку, вы, вероятно, не хотите, чтобы он продолжал повторять попытки бесконечно. Таким образом, Laravel предлагает различные способы указать, сколько раз и как долго может выполняться попытка прослушивания.

Вы можете определить свойство `$tries` в своем классе слушателя, чтобы указать, сколько раз можно попытаться выполнить слушатель, прежде чем он будет считаться неудачным:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Количество попыток слушателя в очереди.
     *
     * @var int
     */
    public $tries = 5;
}
```

В качестве альтернативы определению того, сколько раз можно попытаться выполнить слушатель, прежде чем он потерпит неудачу, вы можете определить время, через которое слушатель больше не должен выполняться. Это позволяет попытаться выполнить прослушивание любое количество раз в течение заданного периода времени. Чтобы определить время, через которое больше не следует предпринимать попытки прослушивания, добавьте метод `retryUntil` в свой класс слушателя. Этот метод должен возвращать экземпляр `DateTime`:

```
use DateTime;
```

```
/**  
 * Определить время, через которое слушатель должен отключиться.  
 *  
 * @return \DateTime  
 */  
public function retryUntil(): DateTime  
{  
    return now()->addMinutes(5);  
}
```

## # Отправка событий

Чтобы отправить событие, вы можете вызвать статический метод `dispatch` события. Этот метод доступен в событии с помощью трейта `Illuminate\Foundation\Events\Dispatchable`. Любые аргументы, переданные методу `dispatch`, будут переданы конструктору события:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Events\OrderShipped;  
use App\Http\Controllers\Controller;  
use App\Models\Order;  
use Illuminate\Http\RedirectResponse;  
use Illuminate\Http\Request;  
  
class OrderShipmentController extends Controller  
{  
    /**  
     * Отправить заказ.  
     */  
    public function store(Request $request): RedirectResponse  
    {  
        $order = Order::findOrFail($request->order_id);  
  
        // Логика отправки заказа ...  
  
        OrderShipped::dispatch($order);  
  
        return redirect('/orders');  
    }  
}
```

Если вы хотите условно отправить событие, вы можете использовать методы `dispatchIf` и `dispatchUnless`:

```
OrderShipped::dispatchIf($condition, $order);
```

```
OrderShipped::dispatchUnless($condition, $order);
```

При тестировании может быть полезным утверждать, что определенные события были отправлены, не активируя их слушателей. В Laravel это легко сделать с помощью [встроенных средств тестирования](#).

## Отправка событий после транзакций в базе данных

Иногда вам может потребоваться указать Laravel отправлять событие только после завершения активной транзакции в базе данных. Для этого вы можете реализовать интерфейс `ShouldDispatchAfterCommit` в классе события.

Этот интерфейс указывает Laravel не отправлять событие, пока текущая транзакция в базе данных не будет завершена. Если транзакция завершится с ошибкой, событие будет отброшено. Если в момент отправки события нет активной транзакции в базе данных, событие будет отправлено немедленно.

```
<?php
```

```
namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Events\ShouldDispatchAfterCommit;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped implements ShouldDispatchAfterCommit
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     *
     */
}
```

```
* Create a new event instance.  
*/  
public function __construct(  
    public Order $order,  
) {}  
}
```

## # Подписчики событий

### Написание подписчиков на события

Подписчики событий – это классы, которые могут подписываться на несколько событий, что позволяет вам определять несколько обработчиков событий в одном классе. Подписчики должны определить метод `subscribe`, которому будет передан экземпляр диспетчера событий. Вы можете вызвать метод `listen` данного диспетчера для регистрации слушателей событий:

```
<?php  
  
namespace App\Listeners;  
  
use Illuminate\Auth\Events\Login;  
use Illuminate\Auth\Events\Logout;  
  
class UserEventSubscriber  
{  
    /**  
     * Обработать событие входа пользователя в систему.  
     */  
    public function handleUserLogin(Login $event): void {}  
  
    /**  
     * Обработать событие выхода пользователя из системы.  
     */  
    public function handleUserLogout(Logout $event): void {}  
  
    /**  
     * Зарегистрировать слушателей для подписчика.  
     *  
     * @param \Illuminate\Events\Dispatcher $events  
     * @return void  
     */  
    public function subscribe(Dispatcher $events): void  
    {
```

```

$events->listen(
    Login::class,
    [UserEventSubscriber::class, 'handleUserLogin']
);

$events->listen(
    Logout::class,
    [UserEventSubscriber::class, 'handleUserLogout']
);
}

}

```

Если ваши методы слушателей событий определены в самом подписчике, вам может быть удобнее возвращать массив событий и имен методов из метода подписчика `subscribe`. Laravel автоматически определит имя класса подписчика при регистрации слушателей событий:

```

<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;

class UserEventSubscriber
{
    /**
     * Обработать событие входа пользователя в систему.
     */
    public function handleUserLogin(Login $event): void {}

    /**
     * Обработать событие выхода пользователя из системы.
     */
    public function handleUserLogout(Logout $event): void {}

    /**
     * Register the listeners for the subscriber.
     */
    public function subscribe(Dispatcher $events): array
    {
        return [
            Login::class => 'handleUserLogin',
            Logout::class => 'handleUserLogout',
        ];
    }
}

```

```
    }  
}
```

## Регистрация подписчиков на события

После написания подписчика Laravel автоматически зарегистрирует методы-обработчики внутри подписчика, если они соответствуют [соглашениям об обнаружении событий](#) Laravel. В противном случае вы можете вручную зарегистрировать своего подписчика, используя метод `subscribe` фасада `Event`. Обычно это следует делать в методе `boot AppServiceProvider` вашего приложения:

```
<?php  
  
namespace App\Providers;  
  
use App\Listeners\UserEventSubscriber;  
use Illuminate\Support\Facades\Event;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Загрузка любых сервисов приложения.  
     */  
    public function boot(): void  
    {  
        Event::subscribe(UserEventSubscriber::class);  
    }  
}
```

## # Тестирование

При тестировании кода, который отправляет события, может потребоваться указать Laravel не выполнять фактически слушателей событий, так как код слушателей можно тестировать непосредственно и отдельно от кода, который отправляет соответствующее событие. Конечно, для тестирования самого слушателя вы можете создать экземпляр слушателя и вызвать метод `handle` напрямую в вашем teste.

Используя метод `fake` фасада `Event`, вы можете предотвратить выполнение слушателей, выполнить код, который требуется протестировать, и затем

утверждать, какие события были отправлены вашим приложением с помощью методов `assertDispatched`, `assertNotDispatched` и `assertNothingDispatched`:

Pest      PHPUnit

<?php

```
use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Support\Facades\Event;

test('orders can be shipped', function () {
    Event::fake();

    // Выполните процесс доставки заказа...

    // Утверждите, что событие было отправлено...
    Event::assertDispatched(OrderShipped::class);

    // Утверждите, что событие было отправлено дважды...
    Event::assertDispatched(OrderShipped::class, 2);

    // Утверждите, что событие не было отправлено...
    Event::assertNotDispatched(OrderFailedToShip::class);

    // Утверждите, что не было отправлено ни одного события...
    Event::assertNothingDispatched();
});
```

Вы можете передать замыкание в методы `assertDispatched` или `assertNotDispatched`, чтобы утверждать, что было отправлено событие, которое проходит заданный "тест истинности". Если хотя бы одно событие было отправлено и прошло заданный тест истинности, то утверждение будет успешным:

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

Если вы хотите просто утверждать, что слушатель события слушает определенное событие, вы можете использовать метод `assertListening`:

```
Event::assertListening(
    OrderShipped::class,
```

```
SendShipmentNotification::class  
);
```

После вызова `Event::fake()`, слушатели событий не будут выполнены. Поэтому, если ваши тесты используют фабрики моделей, которые зависят от событий, например, создание UUID во время события `creating` модели, вы должны вызвать `Event::fake()` **после** использования ваших фабрик.

## Подмена определенного набора событий

Если вы хотите подменить слушателей событий только для определенного набора событий, вы можете передать их в метод `fake` или `fakeFor`:

Pest      PHPUnit

```
test('orders can be processed', function () {  
    Event::fake([
        OrderCreated::class,  
    ]);  
  
    $order = Order::factory()->create();  
  
    Event::assertDispatched(OrderCreated::class);  
  
    // Другие события отправляются как обычно...  
    $order->update([...]);  
});
```

Вы можете подменить все события, кроме указанных событий, используя метод `except`:

```
Event::fake()->except([
    OrderCreated::class,  
]);
```

# Подмена событий в ограниченной области видимости

Если вы хотите подменить слушателей событий только в определенной части вашего теста, вы можете использовать метод `fakeFor`:

Pest      PHPUnit

```
<?php

use App\Events\OrderCreated;
use App\Models\Order;
use Illuminate\Support\Facades\Event;

test('orders can be processed', function () {
    $order = Event::fakeFor(function () {
        $order = Order::factory()->create();

        Event::assertDispatched(OrderCreated::class);

        return $order;
    });

    // Events are dispatched as normal and observers will run ...
    $order->update([...]);
});
```

# Файловое хранилище

## # Введение

## # Конфигурирование

- # Локальный драйвер
- # Публичный диск
- # Предварительная подготовка драйверов
- # Ограниченные и только для чтения файловые системы
- # Файловые системы, совместимые с Amazon S3

## # Доступ к экземплярам дисков

- # Диски по запросу

## # Получение файлов

- # Скачивание файлов
- # URL-адреса файлов
- # Временные URL
- # Метаданные файла

## # Хранение файлов

- # Добавление информации к файлам
- # Копирование и перемещение файлов
- # Автоматическая потоковая передача
- # Загрузка файлов
- # Видимость файла

## # Удаление файлов

## # Каталоги

## # Тестирование

## # Пользовательские файловые системы

## # Введение

Laravel обеспечивает мощную абстракцию файловой системы благодаря замечательному пакету [Flysystem](#) PHP от Фрэнка де Йонга. Интеграция Laravel с Flysystem содержит простые драйверы для работы с локальными файловыми системами, SFTP и Amazon S3. Более того, удивительно просто переключаться между этими вариантами хранения: как локального, так и производственного серверов – поскольку API остается одинаковым для каждой системы.

## # Конфигурирование

Файл конфигурации файловой системы Laravel находится в [config/filesystems.php](#). В этом файле вы можете настроить все «диски» файловой системы. Каждый диск представляет собой определенный драйвер хранилища и место хранения. Примеры конфигураций для каждого поддерживаемого драйвера включены в конфигурационный файл, так что вы можете изменить конфигурацию, отражающую ваши предпочтения хранения и учетные данные.

Драйвер `local` взаимодействует с файлами, хранящимися локально на сервере, на котором запущено приложение Laravel, в то время как драйвер `s3` используется для записи в службу облачного хранилища Amazon S3.

Вы можете настроить столько дисков, сколько захотите, и даже иметь несколько дисков, использующих один и тот же драйвер.

## Локальный драйвер

При использовании драйвера `local` все операции с файлами выполняются относительно корневого каталога, определенного в файле конфигурации `filesystems`. По умолчанию это значение задано каталогом `storage/app`.

Следовательно, следующий метод запишет файл в `storage/app/example.txt`:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('local')->put('example.txt', 'Contents');
```

# Публичный диск

Диск `public`, определенный в файле конфигурации `filesystems` вашего приложения, предназначен для файлов, которые будут общедоступными. По умолчанию публичный диск использует драйвер `local` и хранит свои файлы в `storage/app/public`.

Чтобы сделать эти файлы доступными из интернета, вы должны создать символическую ссылку на `storage/app/public` в `public/storage`. Использование этого соглашения о папках позволит хранить ваши публичные файлы в одном каталоге, который может быть легко доступен между развертываниями при использовании систем развертывания с нулевым временем простоя, таких как [Envoyer](#).

Чтобы создать символическую ссылку, вы можете использовать команду `storage:link` Artisan:

```
php artisan storage:link
```

После того как была создана символическая ссылка, вы можете создавать URL-адреса для сохраненных файлов, используя помощник `asset`:

```
echo asset('storage/file.txt');
```

Вы можете настроить дополнительные символические ссылки в файле конфигурации `filesystems`. Каждая из настроенных ссылок будет создана, когда вы запустите команду `storage:link`:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

Команда `storage:unlink` может быть использована для удаления ваших настроенных символьических ссылок:

```
php artisan storage:unlink
```

## Предварительная подготовка драйверов

## Конфигурирование драйвера S3

Прежде чем начать использовать драйвер S3, вам необходимо установить пакет Flysystem S3 с помощью менеджера пакетов Composer:

```
composer require league/flysystem-aws-s3-v3 "^3.0" --with-all-dependencies
```

Массив конфигурации диска S3 находится в вашем файле конфигурации `config/filesystems.php`. Обычно вам следует настроить информацию и учетные данные S3, используя следующие переменные среды, на которые ссылается файл конфигурации `config/filesystems.php`:

```
AWS_ACCESS_KEY_ID=<your-key-id>
AWS_SECRET_ACCESS_KEY=<your-secret-access-key>
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=<your-bucket-name>
AWS_USE_PATH_STYLE_ENDPOINT=false
```

Для удобства эти переменные среды соответствуют соглашению об именах, используемому в AWS CLI.

## Конфигурирование драйвера FTP

Для использования драйвера FTP, вам нужно установить пакет Flysystem FTP с помощью менеджера пакетов Composer. Выполните следующую команду:

```
composer require league/flysystem-ftp "^3.0"
```

Интеграция Laravel с Flysystem отлично работает с FTP; однако, пример конфигурации по умолчанию не включен в конфигурационный файл `filesystems.php` фреймворка. Если вам нужно настроить файловую систему FTP, вы можете использовать пример конфигурации ниже:

```
'ftp' => [
    'driver' => 'ftp',
    'host' => env('FTP_HOST'),
    'username' => env('FTP_USERNAME'),
    'password' => env('FTP_PASSWORD'),
```

```
// Optional FTP Settings...
// 'port' => env('FTP_PORT', 21),
// 'root' => env('FTP_ROOT'),
// 'passive' => true,
// 'ssl' => true,
// 'timeout' => 30,
],
```

## Конфигурирование драйвера SFTP

Для использования драйвера SFTP вам необходимо установить пакет Flysystem SFTP с помощью менеджера пакетов Composer. Выполните следующую команду:

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Интеграция Laravel с Flysystem отлично работает с SFTP; однако, пример конфигурации по умолчанию не включен в конфигурационный файл `filesystems.php` фреймворка. Если вам нужно настроить файловую систему SFTP, вы можете использовать пример конфигурации ниже:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => env('SFTP_HOST'),

    // Settings for basic authentication...
    'username' => env('SFTP_USERNAME'),
    'password' => env('SFTP_PASSWORD'),

    // Settings for SSH key based authentication with encryption password...
    'privateKey' => env('SFTP_PRIVATE_KEY'),
    'passphrase' => env('SFTP_PASSPHRASE'),

    // Settings for file / directory permissions...
    'visibility' => 'private', // `private` = 0600, `public` = 0644
    'directory_visibility' => 'private', // `private` = 0700, `public` = 0755

    // Optional SFTP Settings...
    // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
    // 'maxTries' => 4,
    // 'passphrase' => env('SFTP_PASSPHRASE'),
    // 'port' => env('SFTP_PORT', 22),
    // 'root' => env('SFTP_ROOT', ''),
    // 'timeout' => 30,
```

```
// 'useAgent' => true,  
],
```

## Ограниченные и только для чтения файловые системы

Ограниченные диски позволяют вам определить файловую систему, в которой все пути автоматически дополняются указанным префиксом пути. Прежде чем создать ограниченный диск файловой системы, вам необходимо установить дополнительный пакет `Flysystem` с помощью менеджера пакетов Composer:

```
composer require league/flysystem-path-prefixing "^3.0"
```

Вы можете создать экземпляр файловой системы с ограниченным путем для любого существующего диска файловой системы, определив диск, который использует драйвер `scoped`. Например, вы можете создать диск, который ограничивает ваш существующий диск `s3` до определенного префикса пути, и затем каждая операция с файлом, использующая ваш ограниченный диск, будет использовать указанный префикс:

```
's3-videos' => [  
    'driver' => 'scoped',  
    'disk' => 's3',  
    'prefix' => 'path/to/videos',  
,
```

“Только для чтения” диски позволяют создавать файловые диски, которые не разрешают операции записи. Прежде чем использовать параметр конфигурации `read-only`, вам необходимо установить дополнительный пакет `Flysystem` с помощью менеджера пакетов Composer:

```
composer require league/flysystem-read-only "^3.0"
```

Затем вы можете включить параметр конфигурации `read-only` в один или несколько массивов конфигурации ваших дисков:

```
's3-videos' => [
    'driver' => 's3',
    // ...
    'read-only' => true,
],
```

## Файловые системы, совместимые с Amazon S3

По умолчанию файл конфигурации вашего приложения `filesystems` содержит конфигурацию диска для диска `s3`. Помимо использования этого диска для взаимодействия с Amazon S3, вы можете использовать его для взаимодействия с любой совместимой с S3 службой хранения файлов, такой как [MinIO](#) или [DigitalOcean Spaces](#).

Обычно после обновления учетных данных диска для соответствия учетным данным службы, которую вы планируете использовать, вам нужно только обновить значение параметра конфигурации `endpoint`. Значение этой опции обычно определяется через переменную окружения `AWS_ENDPOINT`:

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

## MinIO

Для того чтобы интеграция Flysystem в Laravel генерировала правильные URL при использовании MinIO, вам следует определить переменную окружения `AWS_URL`, чтобы она соответствовала локальному URL вашего приложения и включала имя бакета в путь URL:

```
AWS_URL=http://localhost:9000/local
```

Генерация временных URL-адресов для хранилища с использованием метода `temporaryUrl` может не работать при использовании MinIO, если клиент не может получить доступ к конечной точке.

## # Доступ к экземплярам дисков

Фасад `Storage` используется для взаимодействия с любым из ваших сконфигурированных дисков. Например, вы можете использовать метод `put` фасада, чтобы сохранить аватар на диске по умолчанию. Если вы вызываете методы фасада `Storage` без предварительного вызова метода `disk`, то метод будет проксирован на диск по умолчанию:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('avatars/1', $content);
```

Если ваше приложение взаимодействует с несколькими дисками, то вы можете использовать метод `disk` фасада `Storage` для работы с файлами на указанном диске:

```
Storage::disk('s3')->put('avatars/1', $content);
```

## Диски по запросу

Иногда вы можете захотеть создать диск во время выполнения, используя заданную конфигурацию, без того, чтобы эта конфигурация фактически присутствовала в файле конфигурации вашего приложения `filesystems`. Для этого вы можете передать массив конфигурации методу `build` фасада `Storage`:

```
use Illuminate\Support\Facades\Storage;  
  
$disk = Storage::build([  
    'driver' => 'local',  
    'root' => '/path/to/root',  
]);  
  
$disk->put('image.jpg', $content);
```

## # Получение файлов

Метод `get` используется для получения содержимого файла. Необработанное строковое содержимое файла будет возвращено методом. Помните, что все пути к файлам должны быть указаны относительно «корня» диска:

```
$contents = Storage::get('file.jpg');
```

Если файл, который вы извлекаете, содержит JSON, вы можете использовать метод `json` для извлечения файла и декодирования его содержимого:

```
$orders = Storage::json('orders.json');
```

Метод `exists` используется для определения, существует ли файл на диске:

```
if (Storage::disk('s3')->exists('file.jpg')) {  
    // ...  
}
```

Метод `missing` используется, чтобы определить, отсутствует ли файл на диске:

```
if (Storage::disk('s3')->missing('file.jpg')) {  
    // ...  
}
```

## Скачивание файлов

Метод `download` используется для генерации ответа, который заставляет браузер пользователя загружать файл по указанному пути. Метод `download` принимает имя файла в качестве второго аргумента метода, определяющий имя файла, которое видит пользователь, скачивающий этот файл. Наконец, вы можете передать массив заголовков HTTP в качестве третьего аргумента метода:

```
return Storage::download('file.jpg');  
  
return Storage::download('file.jpg', $name, $headers);
```

## URL-адреса файлов

Вы можете использовать метод `url`, чтобы получить URL для указанного файла. Если вы используете драйвер `local`, он обычно просто добавляет `/storage` к

указанному пути и возвращает относительный URL-адрес файла. Если вы используете драйвер `s3`, будет возвращен абсолютный внешний URL-адрес:

```
use Illuminate\Support\Facades\Storage;  
  
$url = Storage::url('file.jpg');
```

При использовании драйвера `local` все файлы, которые должны быть общедоступными, должны быть помещены в каталог `storage/app/public`. Кроме того, вы должны создать символическую ссылку в `public/storage`, которая указывает на каталог `storage/app/public`.

При использовании драйвера `local` возвращаемое значение `url` не является URL-кодированным. По этой причине мы рекомендуем всегда хранить ваши файлы, используя имена, которые будут создавать допустимые URL-адреса.

## Настройка хоста URL

Если вы хотите изменить хост для URL-адресов, созданных с использованием фасада `Storage`, вы можете добавить или изменить параметр `url` в массиве конфигурации диска:

```
'public' => [  
    'driver' => 'local',  
    'root' => storage_path('app/public'),  
    'url' => env('APP_URL').'/storage',  
    'visibility' => 'public',  
    'throw' => false,  
,
```

## Временные URL

Используя метод `temporaryUrl`, вы можете создавать временные URL-адреса для файлов, хранящихся с помощью драйверов `local` и `s3`. Этот метод принимает путь и экземпляр `DateTime`, указывающий, когда должен истечь доступ к файлу по URL:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

## Включение локальных временных URL-адресов

Если вы начали разработку своего приложения до того, как в драйвере `local` появилась поддержка временных URL-адресов, вам может потребоваться включить локальные временные URL-адреса. Для этого добавьте опцию `serve` в массив конфигурации вашего `local` диска в файле конфигурации `config/filesystems.php`:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app/private'),
    'serve' => true, // [tl! add]
    'throw' => false,
],
```

## Параметры запроса S3

Если вам нужно указать дополнительные [параметры запроса S3](#), то вы можете передать массив параметров запроса в качестве третьего аргумент методу `temporaryUrl`:

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
    ]
);
```

## Настройка временных URL-адресов

Если вам нужно настроить способ создания временных URL-адресов для определенного диска хранилища, вы можете использовать метод

`buildTemporaryUrlsUsing`. Например, это может быть полезно, если у вас есть контроллер, позволяющий загружать файлы, хранящиеся на диске, который обычно не поддерживает временные URL-адреса. Обычно этот метод следует вызывать из `boot` метода сервис-провайдера:

```
<?php

namespace App\Providers;

use DateTime;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\URL;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Storage::disk('local')->buildTemporaryUrlsUsing(
            function (string $path, DateTime $expiration, array $options) {
                return URL::temporarySignedRoute(
                    'files.download',
                    $expiration,
                    array_merge($options, ['path' => $path])
                );
            }
        );
    }
}
```

## Временные URL-адреса для загрузки

Возможность генерации временных URL-адресов для загрузки поддерживается только драйвером `s3`.

Если вам нужно создать временный URL-адрес, который можно использовать для загрузки файла непосредственно из вашего клиентского приложения на стороне

клиента, вы можете использовать метод `temporaryUploadUrl`. Этот метод принимает путь и экземпляр `DateTime`, указывающий, когда URL должен истечь. Метод `temporaryUploadUrl` возвращает ассоциативный массив, который можно деструктурировать на URL-адрес для загрузки и заголовки, которые должны включаться в запрос на загрузку:

```
use Illuminate\Support\Facades\Storage;

['url' => $url, 'headers' => $headers] = Storage::temporaryUploadUrl(
    'file.jpg', now()->addMinutes(5)
);
```

Этот метод в основном полезен в серверных средах, где клиентское приложение должно непосредственно загружать файлы в систему облачного хранения, такую как Amazon S3.

## Метаданные файла

Помимо чтения и записи файлов, Laravel также может предоставлять информацию о самих файлах. Например, метод `size` используется для получения размера файла в байтах:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

Метод `lastModified` возвращает временную метку UNIX последнего изменения файла:

```
$time = Storage::lastModified('file.jpg');
```

MIME-тип файла можно получить с помощью метода `mimeType`:

```
$mime = Storage::mimeType('file.jpg');
```

## Пути к файлам

Вы можете использовать метод `path`, чтобы получить путь к указанному файлу. Если вы используете драйвер `local`, он вернет абсолютный путь к файлу. Если вы используете драйвер `s3`, этот метод вернет относительный путь к файлу в корзине `S3`:

```
use Illuminate\Support\Facades\Storage;  
  
$path = Storage::path('file.jpg');
```

## # Хранение файлов

Метод `put` используется для сохранения содержимого файла на диске. Вы также можете передать `resource` PHP методу `put`, который будет использовать поддержку базового потока Flysystem. Помните, что все пути к файлам должны быть указаны относительно «корневого» расположения, настроенного для диска:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('file.jpg', $contents);  
  
Storage::put('file.jpg', $resource);
```

## Обработка ошибок записи

Если метод `put` (или другие операции “записи”) не может записать файл на диск, он вернет `false`:

```
if (!Storage::put('file.jpg', $contents)) {  
    // Файл не удалось записать на диск...  
}
```

По вашему желанию, вы можете определить опцию `throw` в конфигурационном массиве диска вашей файловой системы. Когда эта опция установлена как `true`, методы “записи”, такие как `put`, будут выбрасывать экземпляр `League\Flysystem\UnableToWriteFile`, когда операции записи завершаются неудачей:

```
'public' => [  
    'driver' => 'local',
```

```
// ...
'throw' => true,
],
```

## Добавление информации к файлам

Методы `prepend` и `append` позволяют записывать в начало или конец файла, соответственно:

```
Storage::prepend('file.log', 'Prepended Text');
```

```
Storage::append('file.log', 'Appended Text');
```

## Копирование и перемещение файлов

Метод `copy` используется для копирования существующего файла в новое место на диске, а метод `move` используется для переименования или перемещения существующего файла в новое место:

```
Storage::copy('old/file.jpg', 'new/file.jpg');
```

```
Storage::move('old/file.jpg', 'new/file.jpg');
```

## Автоматическая потоковая передача

Потоковая передача файлов в хранилище позволяет значительно сократить использование памяти. Если вы хотите, чтобы Laravel автоматически управлял потоковой передачей переданного файла в ваше хранилище, вы можете использовать методы `putFile` или `putFileAs`. Эти методы принимают экземпляр `Illuminate\Http\File` или `Illuminate\Http\UploadedFile` и автоматически передают файл в нужное место:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;
```

```
// Автоматически генерировать уникальный идентификатор для имени файла ...
$path = Storage::putFile('photos', new File('/path/to/photo'));
```

```
// Явно указать имя файла ...
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

Следует отметить несколько важных моментов, касающихся метода `putFile`. Обратите внимание, что мы указали только имя каталога, а не имя файла. По умолчанию метод `putFile` генерирует уникальный идентификатор, который будет служить именем файла. Расширение файла будет определено путем проверки MIME-типа файла. Путь к файлу будет возвращен методом `putFile`, так что вы можете сохранить путь, включая сгенерированное имя файла, в вашей базе данных.

Методы `putFile` и `putFileAs` также принимают аргумент для определения «видимости» сохраненного файла. Это особенно полезно, если вы храните файл на облачном диске, таком как Amazon S3, и хотите, чтобы файл был общедоступным через сгенерированные URL:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

## Загрузка файлов

В веб-приложениях одним из наиболее распространенных вариантов хранения файлов является хранение загруженных пользователем файлов, таких как фотографии и документы. Laravel упрощает хранение загруженных файлов с помощью метода `store` экземпляра загружаемого файла. Вызовите метод `store`, указав путь, по которому вы хотите сохранить загруженный файл:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserAvatarController extends Controller
{
    /**
     * Обновить аватар пользователя.
     */
    public function update(Request $request): string
    {
        $path = $request->file('avatar')->store('avatars');
```

```
        return $path;
    }
}
```

В этом примере следует отметить несколько важных моментов. Обратите внимание, что мы указали только имя каталога, а не имя файла. По умолчанию метод `store` генерирует уникальный идентификатор, который будет служить именем файла. Расширение файла будет определено путем проверки MIME-типа файла. Путь к файлу будет возвращен методом `store`, поэтому вы можете сохранить путь, включая сгенерированное имя файла, в своей базе данных.

Вы также можете вызвать метод `putFile` фасада `Storage`, чтобы выполнить ту же операцию сохранения файлов, что и в примере выше:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

## Указание имени файла

Если вы не хотите, чтобы имя файла автоматически присваивалось вашему сохраненному файлу, вы можете использовать метод `storeAs`, который получает путь, имя файла и (необязательный) диск в качестве аргументов:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

Вы также можете использовать метод `putFileAs` фасада `Storage`, который будет выполнять ту же операцию сохранения файлов, что и в примере выше:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

Непечатаемые и недопустимые символы Unicode будут автоматически удалены из путей к файлам. По этой причине, вы *по желанию* можете очистить пути

к файлам перед их передачей в методы хранения файлов Laravel. Пути к файлам нормализуются с помощью метода `League\Flysystem\Util::normalizePath`.

## Указание диска

По умолчанию метод `store` загружаемого файла будет использовать ваш диск по умолчанию. Если вы хотите указать другой диск, передайте имя диска в качестве второго аргумента методу `store`:

```
$path = $request->file('avatar')->store(  
    'avatars' . $request->user()->id, 's3'  
) ;
```

Если вы используете метод `storeAs`, вы можете передать имя диска в качестве третьего аргумента метода:

```
$path = $request->file('avatar')->storeAs(  
    'avatars',  
    $request->user()->id,  
    's3'  
) ;
```

## Другая информация о загружаемом файле

Если вы хотите получить оригинальное имя или расширение загружаемого файла, вы можете сделать это с помощью методов `getClientOriginalName` и `getClientOriginalExtension`:

```
$file = $request->file('avatar');  
  
$name = $file->getClientOriginalName();  
$extension = $file->getClientOriginalExtension();
```

Однако имейте в виду, что методы `getClientOriginalName` и `getClientOriginalExtension` считаются небезопасными, так как имя и расширение файла могут быть изменены

злоумышленником. По этой причине вы обычно должны предпочтеть методы `hashName` и `extension` чтобы получить имя и расширение для загружаемого файла:

```
$file = $request->file('avatar');

$name = $file->hashName(); // Generate a unique, random name...
$extension = $file->extension(); // Determine the file's extension based on the file
```



## Видимость файла

В интеграции Laravel Flysystem «видимость» – это абстракция прав доступа к файлам на нескольких платформах. Файлы могут быть объявлены `public` или `private`. Когда файл объявляется `public`, вы указываете, что файл обычно должен быть доступен для других. Например, при использовании драйвера `s3` вы можете получить URL-адреса для `public` файлов.

Вы можете задать видимость при записи файла с помощью метода `put`:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

Если файл уже был сохранен, его видимость может быть получена и задана с помощью методов `getVisibility` и `setVisibility`, соответственно:

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

При взаимодействии с загружаемыми файлами, вы можете использовать методы `storePublicly` и `storePubliclyAs` для сохранения загружаемого файла с видимостью `public`:

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
```

```
's3'  
);
```

## Локальные файлы и видимость

При использовании драйвера `local`, видимость `public` интерпретируется в право доступа `0755` для каталогов и право доступа `0644` для файлов. Вы можете изменить сопоставление прав доступа в файле конфигурации `filesystems` вашего приложения:

```
'local' => [  
    'driver' => 'local',  
    'root' => storage_path('app'),  
    'permissions' => [  
        'file' => [  
            'public' => 0644,  
            'private' => 0600,  
        ],  
        'dir' => [  
            'public' => 0755,  
            'private' => 0700,  
        ],  
    ],  
    'throw' => false,  
,
```

## # Удаление файлов

Метод `delete` принимает имя одного файла или массив имен файлов для удаления:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::delete('file.jpg');  
  
Storage::delete(['file.jpg', 'file2.jpg']);
```

При необходимости вы можете указать диск, с которого следует удалить файл:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('s3')->delete('path/file.jpg');
```

# # Каталоги

## Получение всех файлов каталога

Метод `files` возвращает массив всех файлов указанного каталога. Если вы хотите получить список всех файлов каталога, включая все подкаталоги, вы можете использовать метод `allFiles`:

```
use Illuminate\Support\Facades\Storage;  
  
$files = Storage::files($directory);  
  
$files = Storage::allFiles($directory);
```

## Получение всех каталогов из каталога

Метод `directories` возвращает массив всех каталогов указанного каталога. Кроме того, вы можете использовать метод `allDirectories`, чтобы получить список всех каталогов внутри указанного каталога и всех его подкаталогов:

```
$directories = Storage::directories($directory);  
  
$directories = Storage::allDirectories($directory);
```

## Создание каталога

Метод `makeDirectory` создаст указанный каталог, включая все необходимые подкаталоги:

```
Storage::makeDirectory($directory);
```

## Удаление каталога

Наконец, для удаления каталога и всех его файлов можно использовать метод `deleteDirectory`:

```
Storage::deleteDirectory($directory);
```

# # Тестирование

Метод `fake` фасада `Storage` позволяет вам легко создавать фейковый диск, который, в сочетании с утилитами генерации файлов класса `Illuminate\Http\UploadedFile`, значительно упрощает тестирование загрузки файлов. Например:

Pest      PHPUnit

```
<?php

use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;

test('albums can be uploaded', function () {
    Storage::fake('photos');

    $response = $this->json('POST', '/photos', [
        UploadedFile::fake()->image('photo1.jpg'),
        UploadedFile::fake()->image('photo2.jpg')
    ]);

    // Assert one or more files were stored...
    Storage::disk('photos')->assertExists('photo1.jpg');
    Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

    // Assert one or more files were not stored...
    Storage::disk('photos')->assertMissing('missing.jpg');
    Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);

    // Assert that a given directory is empty...
    Storage::disk('photos')->assertDirectoryEmpty('/wallpapers');
});
```

По умолчанию метод `fake` будет удалять все файлы в своей временной директории. Если вы хотите сохранить эти файлы, вы можете вместо этого использовать метод `"persistentFake"`. Для получения дополнительной информации о тестировании загрузки файлов вы можете проконсультироваться с [документацией по тестированию HTTP, касающейся загрузки файлов](#).

Метод `image` требует наличие [расширения GD](#).

# # Пользовательские файловые системы

Интеграция Laravel с Flysystem обеспечивает поддержку нескольких «драйверов» из коробки; однако, Flysystem этим не ограничивается и имеет адаптеры для многих других систем хранения. Вы можете создать собственный драйвер, если хотите использовать один из этих дополнительных адаптеров в своем приложении Laravel.

Чтобы определить собственную файловую систему, вам понадобится адаптер Flysystem. Давайте добавим в наш проект адаптер Dropbox, поддерживаемый сообществом:

```
composer require spatie/flysystem-dropbox
```

Затем вы можете зарегистрировать драйвер в методе `boot` одного из [поставщиков служб](#) вашего приложения. Для этого вы должны использовать метод `extend` фасада `Storage`:

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Foundation\Application;
use Illuminate\Filesystem\FilesystemAdapter;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     */
}
```

```
public function boot(): void
{
    Storage::extend('dropbox', function (Application $app, array $config) {
        $adapter = new DropboxAdapter(new DropboxClient(
            $config['authorization_token']
        ));

        return new FilesystemAdapter(
            new Filesystem($adapter, $config),
            $adapter,
            $config
        );
    });
}
```

Первый аргумент метода `extend` – это имя драйвера, а второй – замыкание, которое получает переменные `$app` и `$config`. Замыкание должно возвращать экземпляр `Illuminate\Filesystem\FilesystemAdapter`. Переменная `$config` содержит значения, определенные в `config/filesystems.php` для указанного диска.

После того как вы создали и зарегистрировали расширение поставщика службы, вы можете использовать драйвер `dropbox` в вашем файле конфигурации `config/filesystems.php`.

# Глобальные помощники (helpers)

# Введение

# Доступные методы

# Массивы и объекты

# Числа

# Пути

# URL-адреса

# Разное

# Массивы и объекты

# Числа

# Пути

# URL-адреса

# Разное

# Другие утилиты

# Benchmark

# Даты

# Отложенные функции

# Лотерея

# Pipeline

# Sleep

## # Введение

Laravel содержит множество глобальных «вспомогательных» функций. Многие из этих функций используются самим фреймворком; однако, вы можете использовать их в своих собственных приложениях, если сочтете удобными.

## # Доступные методы

# Массивы и объекты

[Arr::accessible](#)

[Arr::add](#)

[Arr::collapse](#)

[Arr::crossJoin](#)

[Arr::divide](#)

[Arr::dot](#)

[Arr::except](#)

[Arr::exists](#)

[Arr::first](#)

[Arr::flatten](#)

[Arr::forget](#)

[Arr::get](#)

[Arr::has](#)

[Arr::hasAny](#)

[Arr::isAssoc](#)

[Arr::isList](#)

[Arr::join](#)

[Arr::keyBy](#)

[Arr::last](#)

[Arr::map](#)

[Arr::mapSpread](#)

[Arr::mapWithKeys](#)

[Arr::only](#)

[Arr::pluck](#)

[Arr::prepend](#)

[Arr::prependKeysWith](#)

[Arr::pull](#)

[Arr::query](#)

[Arr::random](#)

[Arr::set](#)

[Arr::shuffle](#)

[Arr::sort](#)

[Arr::sortDesc](#)

[Arr::sortRecursive](#)

[Arr::take](#)  
[Arr::toCssClasses](#)  
[Arr::toCssStyles](#)  
[Arr::undot](#)  
[Arr::where](#)  
[Arr::whereNotNull](#)  
[Arr::wrap](#)  
[data\\_fill](#)  
[data\\_get](#)  
[data\\_set](#)  
[data\\_forget](#)  
[head](#)  
[last](#)

## Числа

[Number::abbreviate](#)  
[Number::clamp](#)  
[Number::currency](#)  
[Number::defaultCurrency](#)  
[Number::defaultLocale](#)  
[Number::fileSize](#)  
[Number::forHumans](#)  
[Number::format](#)  
[Number::ordinal](#)  
[Number::pairs](#)  
[Number::percentage](#)  
[Number::spell](#)  
[Number::trim](#)  
[Number::useLocale](#)  
[Number::withLocale](#)  
[Number::useCurrency](#)  
[Number::withCurrency](#)

## Пути

app\_path  
base\_path  
config\_path  
database\_path  
lang\_path  
mix  
public\_path  
resource\_path  
storage\_path

## URL-адреса

action  
asset  
route  
secure\_asset  
secure\_url  
to\_route  
url

## Разное

abort  
abort\_if  
abort\_unless  
app  
auth  
back  
bcrypt  
blank  
broadcast  
cache  
class\_uses\_recursive  
collect  
config  
context  
cookie

[csrf\\_field](#)  
[csrf\\_token](#)  
[decrypt](#)  
[dd](#)  
[dispatch](#)  
[dispatch\\_sync](#)  
[dump](#)  
[encrypt](#)  
[env](#)  
[event](#)  
[fake](#)  
[filled](#)  
[info](#)  
[literal](#)  
[logger](#)  
[method\\_field](#)  
[now](#)  
[old](#)  
[once](#)  
[optional](#)  
[policy](#)  
[redirect](#)  
[report](#)  
[report\\_if](#)  
[report\\_unless](#)  
[request](#)  
[rescue](#)  
[resolve](#)  
[response](#)  
[retry](#)  
[session](#)  
[tap](#)  
[throw\\_if](#)  
[throw\\_unless](#)  
[today](#)  
[trait\\_uses\\_recursive](#)

[transform](#)

[validator](#)

[value](#)

[view](#)

[with](#)

[when](#)

## # Массивы и объекты

### Arr::accessible()

Метод `Arr::accessible` определяет, доступно ли переданное значение массиву:

```
use Illuminate\Support\Arr;
use Illuminate\Support\Collection;

$isAccessible = Arr::accessible(['a' => 1, 'b' => 2]);
// true

$isAccessible = Arr::accessible(new Collection);
// true

$isAccessible = Arr::accessible('abc');
// false

$isAccessible = Arr::accessible(new stdClass);
// false
```

### Arr::add()

Метод `Arr::add` добавляет переданную пару ключ / значение в массив, если указанный ключ еще не существует в массиве или установлен как `null`:

```
use Illuminate\Support\Arr;

$array = Arr::add(['name' => 'Desk'], 'price', 100);
```

```
// ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

## Arr::collapse()

Метод `Arr::collapse` сворачивает массив массивов в один массив:

```
use Illuminate\Support\Arr;

$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Arr::crossJoin()

Метод `Arr::crossJoin` перекрестно соединяет указанные массивы, возвращая декартово произведение со всеми возможными перестановками:

```
use Illuminate\Support\Arr;

$matrix = Arr::crossJoin([1, 2], ['a', 'b']);

/*
[
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
]
*/

$matrix = Arr::crossJoin([1, 2], ['a', 'b'], ['I', 'II']);

/*
[
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
]
```

```
[2, 'b', 'I'],
[2, 'b', 'II'],
]
*/
```

## Arr::divide()

Метод `Arr::divide` возвращает два массива: один содержит ключи, а другой – значения переданного массива:

```
use Illuminate\Support\Arr;

[$keys, $values] = Arr::divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

## Arr::dot()

Метод `Arr::dot` объединяет многомерный массив в одноуровневый, использующий «точечную нотацию» для обозначения глубины:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$flattened = Arr::dot($array);

// ['products.desk.price' => 100]
```

## Arr::except()

Метод `Arr::except` удаляет переданные пары ключ / значение из массива:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$filtered = Arr::except($array, ['price']);
```

```
// ['name' => 'Desk']
```

## Arr::exists()

Метод `Arr::exists` проверяет, существует ли переданный ключ в указанном массиве:

```
use Illuminate\Support\Arr;

$array = ['name' => 'John Doe', 'age' => 17];

$exists = Arr::exists($array, 'name');

// true

$exists = Arr::exists($array, 'salary');

// false
```

## Arr::first()

Метод `Arr::first` возвращает первый элемент массива, прошедший тест переданного замыкания на истинность:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300];

$first = Arr::first($array, function (int $value, int $key) {
    return $value >= 150;
});

// 200
```

Значение по умолчанию может быть передано в качестве третьего аргумента методу. Это значение будет возвращено, если ни одно из значений не пройдет проверку на истинность:

```
use Illuminate\Support\Arr;
```

```
$first = Arr::first($array, $callback, $default);
```

## Arr::flatten()

Метод `Arr::flatten` объединяет многомерный массив в одноуровневый:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$flattened = Arr::flatten($array);

// ['Joe', 'PHP', 'Ruby']
```

## Arr::forget()

Метод `Arr::forget` удаляет переданную пару ключ / значение из глубоко вложенного массива, используя «точечную нотацию»:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk');

// ['products' => []]
```

## Arr::get()

Метод `Arr::get` извлекает значение из глубоко вложенного массива, используя «точечную нотацию»:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price');

// 100
```

Метод `Arr::get` также принимает значение по умолчанию, которое будет возвращено, если указанный ключ отсутствует в массиве:

```
use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0);

// 0
```

## Arr::has()

Метод `Arr::has` проверяет, существует ли переданный элемент или элементы в массиве, используя «точечную нотацию»:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::has($array, 'product.name');

// true

$contains = Arr::has($array, ['product.price', 'product.discount']);

// false
```

## Arr::hasAny()

Метод `Arr::hasAny` проверяет, существует ли какой-либо элемент в переданном наборе в массиве, используя «точечную нотацию»:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::hasAny($array, 'product.name');

// true

$contains = Arr::hasAny($array, ['product.name', 'product.discount']);

// true
```

```
$contains = Arr::hasAny($array, ['category', 'product.discount']);  
// false
```

## Arr::isAssoc()

Метод `Arr::isAssoc` возвращает `true`, если переданный массив является ассоциативным. Массив считается ассоциативным, если в нем нет последовательных цифровых ключей, начинающихся с нуля:

```
use Illuminate\Support\Arr;  
  
$isAssoc = Arr::isAssoc(['product' => ['name' => 'Desk', 'price' => 100]]);  
// true  
  
$isAssoc = Arr::isAssoc([1, 2, 3]);  
// false
```

## Arr::isList()

Метод `Arr::isList` возвращает `true`, если ключи заданного массива представляют собой последовательные целые числа, начиная с нуля:

```
use Illuminate\Support\Arr;  
  
$isList = Arr::isList(['foo', 'bar', 'baz']);  
// true  
  
$isList = Arr::isList(['product' => ['name' => 'Desk', 'price' => 100]]);  
// false
```

## Arr::join()

Метод `Arr::join` объединяет элементы массива в строку. Используя второй аргумента этого метода вы также можете указать строку для соединения последнего элемента массива:

```
use Illuminate\Support\Arr;

$array = ['Tailwind', 'Alpine', 'Laravel', 'Livewire'];

$joined = Arr::join($array, ', ');

// Tailwind, Alpine, Laravel, Livewire

$joined = Arr::join($array, ', ', ' and ');

// Tailwind, Alpine, Laravel and Livewire
```

## Arr::keyBy()

Метод `Arr::keyBy` присваивает ключи элементам базового массива на основе указанного ключа. Если у нескольких элементов один и тот же ключ, в новом массиве появится только последний:

```
use Illuminate\Support\Arr;

$array = [
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
];

$keyed = Arr::keyBy($array, 'product_id');

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
```

## Arr::last()

Метод `Arr::last` возвращает последний элемент массива, прошедший тест переданного замыкания на истинность:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300, 110];
```

```
$last = Arr::last($array, function (int $value, int $key) {
    return $value >= 150;
});

// 300
```

Значение по умолчанию может быть передано в качестве третьего аргумента методу. Это значение будет возвращено, если ни одно из значений не пройдет проверку на истинность:

```
use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);
```

## Arr::map()

Метод `Arr::map` проходит по массиву и передает каждое значение и ключ указанной функции обратного вызова. Значение массива заменяется значением, возвращаемым обратным вызовом:

```
use Illuminate\Support\Arr;

$array = ['first' => 'james', 'last' => 'kirk'];

$mapped = Arr::map($array, function (string $value, string $key) {
    return ucfirst($value);
});

// ['first' => 'James', 'last' => 'Kirk']
```

## Arr::mapSpread()

Метод `Arr::mapSpread` выполняет итерацию по массиву, передавая каждое значение вложенного элемента в данное замыкание. Замыкание может изменять элемент и возвращать его, формируя таким образом новый массив измененных элементов:

```
use Illuminate\Support\Arr;

$array = [
    [0, 1],
    [2, 3],
```

```

[4, 5],
[6, 7],
[8, 9],
];

$mapped = Arr::mapSpread($array, function (int $even, int $odd) {
    return $even + $odd;
});

/*
[1, 5, 9, 13, 17]
*/

```

## Arr::mapWithKeys()

Метод `Arr::mapWithKeys` проходит по массиву и передает каждое значение указанной функции обратного вызова, которая должна возвращать ассоциативный массив, содержащий одну пару ключ / значение:

```
use Illuminate\Support\Arr;
```

```

$array = [
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
];

$mapped = Arr::mapWithKeys($array, function (array $item, int $key) {
    return [$item['email'] => $item['name']];
});

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/

```

## Arr::only()

Метод `Arr::only` возвращает только указанные пары ключ / значение из переданного массива:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

## Arr::pluck()

Метод `Arr::pluck` извлекает все значения для указанного ключа из массива:

```
use Illuminate\Support\Arr;

$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$names = Arr::pluck($array, 'developer.name');

// ['Taylor', 'Abigail']
```

Вы также можете задать ключ результирующего списка:

```
use Illuminate\Support\Arr;

$names = Arr::pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail']
```

## Arr::prepend()

Метод `Arr::prepend` помещает элемент в начало массива:

```
use Illuminate\Support\Arr;

$array = ['one', 'two', 'three', 'four'];

$array = Arr::prepend($array, 'zero');

// ['zero', 'one', 'two', 'three', 'four']
```

При необходимости вы можете указать ключ, который следует использовать для значения:

```
use Illuminate\Support\Arr;

$array = ['price' => 100];

$array = Arr::prepend($array, 'Desk', 'name');

// ['name' => 'Desk', 'price' => 100]
```

## Arr::prependKeysWith()

Метод `Arr::prependKeysWith` добавляет указанный префикс ко всем именам ключей ассоциативного массива:

```
use Illuminate\Support\Arr;

$array = [
    'name' => 'Desk',
    'price' => 100,
];

$keyed = Arr::prependKeysWith($array, 'product.');

/*
[
    'product.name' => 'Desk',
    'product.price' => 100,
]
*/
```

## Arr::pull()

Метод `Arr::pull` возвращает и удаляет пару ключ / значение из массива:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$name = Arr::pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

Значение по умолчанию может быть передано в качестве третьего аргумента методу. Это значение будет возвращено, если ключ не существует:

```
use Illuminate\Support\Arr;

$value = Arr::pull($array, $key, $default);
```

## Arr::query()

Метод `Arr::query` преобразует массив в строку запроса:

```
use Illuminate\Support\Arr;

$array = [
    'name' => 'Taylor',
    'order' => [
        'column' => 'created_at',
        'direction' => 'desc'
    ]
];

Arr::query($array);

// name=Taylor&order[column]=created_at&order[direction]=desc
```

## Arr::random()

Метод `Arr::random` возвращает случайное значение из массива:

```
use Illuminate\Support\Arr;

$array = [1, 2, 3, 4, 5];

$random = Arr::random($array);

// 4 - (retrieved randomly)
```

Вы также можете указать количество элементов для возврата в качестве необязательного второго аргумента. Обратите внимание, что при указании этого аргумента, будет возвращен массив, даже если требуется только один элемент:

```
use Illuminate\Support\Arr;

$item = Arr::random($array, 2);

// [2, 5] - (retrieved randomly)
```

## Arr::set()

Метод `Arr::set` устанавливает значение с помощью «точечной нотации» во вложенном массиве:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

## Arr::shuffle()

Метод `Arr::shuffle` случайным образом перемешивает элементы в массиве:

```
use Illuminate\Support\Arr;

$array = Arr::shuffle([1, 2, 3, 4, 5]);

// [3, 2, 5, 1, 4] - (generated randomly)
```

## Arr::sort()

Метод `Arr::sort` сортирует массив по его значениям:

```
use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sort($array);

// ['Chair', 'Desk', 'Table']
```

Вы также можете отсортировать массив по результатам переданного замыкания:

```
use Illuminate\Support\Arr;

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(Arr::sort($array, function (array $value) {
    return $value['name'];
}));
```

/\*  
[  
 ['name' => 'Chair'],  
 ['name' => 'Desk'],  
 ['name' => 'Table'],  
]  
\*/

## Arr::sortDesc()

Метод `Arr::sortDesc` сортирует массив по убыванию значений:

```
use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sortDesc($array);
```

```
// ['Table', 'Desk', 'Chair']
```

Вы также можете отсортировать массив по результатам переданного замыкания:

```
use Illuminate\Support\Arr;

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(Arr::sortDesc($array, function (array $value) {
    return $value['name'];
}));
```

/\*  
 \* [  
 \* ['name' => 'Table'],  
 \* ['name' => 'Desk'],  
 \* ['name' => 'Chair'],  
 \* ]  
 \*/

## Arr::sortRecursive()

Метод `Arr::sortRecursive` рекурсивно сортирует массив с помощью метода `sort` для числовых подмассивов и `ksort` для ассоциативных подмассивов:

```
use Illuminate\Support\Arr;

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
    ['one' => 1, 'two' => 2, 'three' => 3],
];

$sorted = Arr::sortRecursive($array);

/*
 * [
 *     ['JavaScript', 'PHP', 'Ruby'],
 *     ['one' => 1, 'three' => 3, 'two' => 2],
 *     ['Li', 'Roman', 'Taylor'],
 * ]
```

```
]  
*/
```

Если вы хотите, чтобы результаты были отсортированы по убыванию, вы можете использовать метод `Arr::sortRecursiveDesc`.

```
$sorted = Arr::sortRecursiveDesc($array);
```

## Arr::take() {.collection-method}

Метод `Arr::take` возвращает новый массив с указанным количеством элементов:

```
use Illuminate\Support\Arr;  
  
$array = [0, 1, 2, 3, 4, 5];  
  
$chunk = Arr::take($array, 3);  
  
// [0, 1, 2]
```

Вы также можете передать отрицательное целое число, чтобы получить указанное количество элементов с конца массива:

```
$array = [0, 1, 2, 3, 4, 5];  
  
$chunk = Arr::take($array, -2);  
  
// [4, 5]
```

## Arr::toCssClasses()

Метод `Arr::toCssClasses` составляет строку классов CSS исходя из заданных условий. Метод принимает массив классов, где ключ массива содержит класс или классы, которые вы хотите добавить, а значение является булевым выражением. Если элемент массива не имеет строкового ключа, он всегда будет включен в список отрисованных классов:

```
use Illuminate\Support\Arr;
```

```
$isActive = false;
$hasError = true;

$array = ['p-4', 'font-bold' => $isActive, 'bg-red' => $hasError];

$classes = Arr::toCssClasses($array);

/*
 'p-4 bg-red'
*/
```

## Arr::toCssStyles()

Метод `Arr::toCssStyles` условно компилирует строку стилей CSS. Метод принимает массив классов, где ключ массива содержит класс или классы, которые вы хотите добавить, а значение – логическое выражение. Если элемент массива имеет числовой ключ, он всегда будет включен в список отображаемых классов:

```
use Illuminate\Support\Arr;

$hasColor = true;

$array = ['background-color: blue', 'color: blue' => $hasColor];

$classes = Arr::toCssStyles($array);

/*
 'background-color: blue; color: blue;'
*/
```

При помощи этого метода осуществляется [объединение css-классов в Blade](#), а также [в директиве @class](#).

## Arr::undot()

Метод `Arr::undot` расширяет одномерный массив, использующий “точечную нотацию”, в многомерный массив:

```
use Illuminate\Support\Arr;

$array = [
    'user.name' => 'Kevin Malone',
    'user.occupation' => 'Accountant',
```

```
];
$array = Arr::undot($array);

// ['user' => ['name' => 'Kevin Malone', 'occupation' => 'Accountant']]
```

## Arr::where()

Метод `Arr::where` фильтрует массив, используя переданное замыкание:

```
use Illuminate\Support\Arr;

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function (string|int $value, int $key) {
    return is_string($value);
});

// [1 => '200', 3 => '400']
```

## Arr::whereNotNull()

Метод `Arr::whereNotNull` удаляет все значения `null` из данного массива:

```
use Illuminate\Support\Arr;

$array = [0, null];

$filtered = Arr::whereNotNull($array);

// [0 => 0]
```

## Arr::wrap()

Метод `Arr::wrap` оборачивает переданное значение в массив. Если переданное значение уже является массивом, то оно будет возвращено без изменений:

```
use Illuminate\Support\Arr;

$string = 'Laravel';

$array = Arr::wrap($string);
```

```
// ['Laravel']
```

Если переданное значение равно `null`, то будет возвращен пустой массив:

```
use Illuminate\Support\Arr;  
  
$array = Arr::wrap(null);  
  
// []
```

## data\_fill()

Функция `data_fill` устанавливает отсутствующее значение с помощью «точечной нотации» во вложенном массиве или объекте:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_fill($data, 'products.desk.price', 200);  
  
// ['products' => ['desk' => ['price' => 100]]]  
  
data_fill($data, 'products.desk.discount', 10);  
  
// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]
```

Допускается использование метасимвола подстановки `*`:

```
$data = [  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 100],  
        ['name' => 'Desk 2'],  
    ],  
];  
  
data_fill($data, 'products.*.price', 200);  
  
/*  
[  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 100],  
        ['name' => 'Desk 2', 'price' => 200],  
    ],
```

```
]  
*/
```

## data\_get()

Функция `data_get` возвращает значение с помощью «точечной нотации» из вложенного массива или объекта:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
$price = data_get($data, 'products.desk.price');  
  
// 100
```

Функция `data_get` также принимает значение по умолчанию, которое будет возвращено, если указанный ключ не найден:

```
$discount = data_get($data, 'products.desk.discount', 0);  
  
// 0
```

Допускается использование метасимвола подстановки `*`, предназначенный для любого ключа массива или объекта:

```
$data = [  
    'product-one' => ['name' => 'Desk 1', 'price' => 100],  
    'product-two' => ['name' => 'Desk 2', 'price' => 150],  
];  
  
data_get($data, '* .name');  
  
// ['Desk 1', 'Desk 2'];
```

Заполнители `{first}` и `{last}` могут использоваться для получения первого или последнего элемента массива:

```
$flight = [  
    'segments' => [  
        ['from' => 'LHR', 'departure' => '9:00', 'to' => 'IST', 'arrival' => '15:00'],  
        ['from' => 'IST', 'departure' => '16:00', 'to' => 'PKX', 'arrival' => '20:00']
```

```
],
];

data_get($flight, 'segments.{first}.arrival');

// 15:00
```

## data\_set()

Функция `data_set` устанавливает значение с помощью «точечной нотации» во вложенном массиве или объекте:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

Допускается использование метасимвола подстановки `*`:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 150],
    ],
];

data_set($data, 'products.*.price', 200);

/*
[
    'products' => [
        ['name' => 'Desk 1', 'price' => 200],
        ['name' => 'Desk 2', 'price' => 200],
    ],
]
```

По умолчанию все существующие значения перезаписываются. Если вы хотите, чтобы значение было установлено только в том случае, если оно не существует, вы можете передать `false` в качестве четвертого аргумента:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_set($data, 'products.desk.price', 200, overwrite: false);  
  
// ['products' => ['desk' => ['price' => 200]]]
```

## data\_forget()

Функция `data_forget` удаляет значение внутри вложенного массива или объекта, используя “точечную” нотацию:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_forget($data, 'products.desk.price');  
  
// ['products' => ['desk' => []]]
```

Эта функция также принимает маски с использованием звездочек и удаляет соответствующие значения из цели:

```
$data = [  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 100],  
        ['name' => 'Desk 2', 'price' => 150],  
    ],  
];  
  
data_forget($data, 'products.*.price');  
  
/*  
[  
    'products' => [  
        ['name' => 'Desk 1'],  
        ['name' => 'Desk 2'],  
    ],  
]
```

## head()

Функция `head` возвращает первый элемент переданного массива:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

## last()

Функция `last` возвращает последний элемент переданного массива:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

## # Числа

### Number::abbreviate()

Метод `Number::abbreviate` возвращает числовое значение в удобочитаемом формате с сокращением для единиц измерения:

```
use Illuminate\Support\Number;

$number = Number::abbreviate(1000);

// 1K

$number = Number::abbreviate(489939);

// 490K

$number = Number::abbreviate(1230000, precision: 2);

// 1.23M
```

### Number::clamp() {.collection-method}

Метод `Number::clamp` гарантирует, что заданное число останется в заданном диапазоне. Если число меньше минимума, возвращается минимальное значение. Если число больше максимума, возвращается максимальное значение:

```
use Illuminate\Support\Number;

$number = Number::clamp(105, min: 10, max: 100);

// 100

$number = Number::clamp(5, min: 10, max: 100);

// 10

$number = Number::clamp(10, min: 10, max: 100);

// 10

$number = Number::clamp(20, min: 10, max: 100);

// 20
```

## Number::currency()

Метод `Number::currency` возвращает представление указанного значения в валюте в виде строки:

```
use Illuminate\Support\Number;

$currency = Number::currency(1000);

// $1,000.00

$currency = Number::currency(1000, in: 'EUR');

// €1,000.00

$currency = Number::currency(1000, in: 'EUR', locale: 'de');

// 1.000,00 €
```

## Number::defaultCurrency()

Метод `Number::defaultCurrency` возвращает валюту по умолчанию, используемую классом `Number`:

```
use Illuminate\Support\Number;

$currency = Number::defaultCurrency();

// USD
```

## Number::defaultLocale()

Метод `Number::defaultLocale` возвращает локаль по умолчанию, используемую классом `Number`:

```
use Illuminate\Support\Number;

$locale = Number::defaultLocale();

// en
```

## Number::fileSize()

Метод `Number::fileSize` для указанного значения в байтах возвращает представление размера файла в виде строки:

```
use Illuminate\Support\Number;

$size = Number::fileSize(1024);

// 1 KB

$size = Number::fileSize(1024 * 1024);

// 1 MB

$size = Number::fileSize(1024, precision: 2);

// 1.00 KB
```

## Number::forHumans()

Метод `Number::forHumans` возвращает числовое значение в удобочитаемом формате:

```
use Illuminate\Support\Number;

$number = Number::forHumans(1000);

// 1 thousand

$number = Number::forHumans(489939);

// 490 thousand

$number = Number::forHumans(1230000, precision: 2);

// 1.23 million
```

## Number::format()

Метод `Number::format` форматирует предоставленное число в строку с учетом локализации:

```
use Illuminate\Support\Number;

$number = Number::format(100000);

// 100,000

$number = Number::format(100000, precision: 2);

// 100,000.00

$number = Number::format(100000.123, maxPrecision: 2);

// 100,000.12

$number = Number::format(100000, locale: 'de');

// 100.000
```

## Number::ordinal() {.collection-method}

Метод `Number::ordinal` возвращает порядковое представление числа:

```
use Illuminate\Support\Number;

$number = Number::ordinal(1);

// 1st

$number = Number::ordinal(2);

// 2nd

$number = Number::ordinal(21);

// 21st
```

## Number::pairs()

Метод `Number::pairs` генерирует массив пар чисел (поддиапазонов) на основе указанного диапазона и значения шага. Этот метод может быть полезен для разделения большего диапазона чисел на более мелкие, управляемые поддиапазоны для таких задач, как разбивка на страницы или пакетная обработка. Метод `pairs` возвращает массив массивов, где каждый внутренний массив представляет пару (поддиапазон) чисел:

```
use Illuminate\Support\Number;

$result = Number::pairs(25, 10);

// [[1, 10], [11, 20], [21, 25]]

$result = Number::pairs(25, 10, offset: 0);

// [[0, 10], [10, 20], [20, 25]]
```

## Number::percentage()

Метод `Number::percentage` возвращает процентное представление указанного значения в виде строки:

```
use Illuminate\Support\Number;

$percentage = Number::percentage(10);
```

```
// 10%  
  
$percentage = Number::percentage(10, precision: 2);  
  
// 10.00%  
  
$percentage = Number::percentage(10.123, maxPrecision: 2);  
  
// 10.12%  
  
$percentage = Number::percentage(10, precision: 2, locale: 'de');  
  
// 10,00%
```

## Number::spell() {.collection-method}

Метод `Number::spell` возвращает заданное число прописью:

```
use Illuminate\Support\Number;  
  
$number = Number::spell(102);  
  
// one hundred and two  
  
$number = Number::spell(88, locale: 'fr');  
  
// quatre-vingt-huit
```

Аргумент `after` позволяет указать значение, после которого все числа должны быть прописью:

```
$number = Number::spell(10, after: 10);  
  
// 10  
  
$number = Number::spell(11, after: 10);  
  
// eleven
```

Аргумент `until` позволяет указать значение, до которого все числа должны быть прописью:

```
$number = Number::spell(5, until: 10);  
  
// five  
  
$number = Number::spell(10, until: 10);  
  
// 10
```

## Number::trim()

Метод `Number::trim` удаляет все конечные нулевые цифры после десятичной точки заданного числа:

```
use Illuminate\Support\Number;  
  
$number = Number::trim(12.0);  
  
// 12  
  
$number = Number::trim(12.30);  
  
// 12.3
```

## Number::useLocale() {.collection-method}

Метод `Number::useLocale` глобально устанавливает языковой стандарт чисел по умолчанию, что влияет на форматирование чисел и валюты при последующих обращениях к методам класса `Number`:

```
use Illuminate\Support\Number;  
  
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void  
{  
    Number::useLocale('de');  
}
```

## Number::withLocale() {.collection-method}

Метод `Number::withLocale` выполняет заданное замыкание с использованием указанного языкового стандарта, а затем восстанавливает исходный языковой стандарт после выполнения замыкания:

```
use Illuminate\Support\Number;

$number = Number::withLocale('de', function () {
    return Number::format(1500);
});
```

## Number::useCurrency()

Метод `Number::useCurrency` устанавливает глобальную числовую валюту по умолчанию, что влияет на форматирование валюты при последующих вызовах методов класса `Number`:

```
use Illuminate\Support\Number;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Number::useCurrency('GBP');
}
```

## Number::withCurrency()

Метод `Number::withCurrency` выполняет данное замыкание, используя указанную валюту, а затем восстанавливает исходную валюту после выполнения обратного вызова:

```
use Illuminate\Support\Number;

$number = Number::withCurrency('GBP', function () {
    // ...
});
```

# # Пути

## app\_path()

Функция `app_path` возвращает полный путь к каталогу вашего приложения `app`. Вы также можете использовать функцию `app_path` для создания полного пути к файлу относительно каталога приложения:

```
$path = app_path();  
  
$path = app_path('Http/Controllers/Controller.php');
```

## base\_path()

Функция `base_path` возвращает полный путь к корневому каталогу вашего приложения. Вы также можете использовать функцию `base_path` для генерации полного пути к заданному файлу относительно корневого каталога проекта:

```
$path = base_path();  
  
$path = base_path('vendor/bin');
```

## config\_path()

Функция `config_path` возвращает полный путь к каталогу `config` вашего приложения. Вы также можете использовать функцию `config_path` для создания полного пути к заданному файлу в каталоге конфигурации приложения:

```
$path = config_path();  
  
$path = config_path('app.php');
```

## database\_path()

Функция `database_path` возвращает полный путь к каталогу `database` вашего приложения. Вы также можете использовать функцию `database_path` для генерации полного пути к заданному файлу в каталоге базы данных:

```
$path = database_path();
```

```
$path = database_path('factories/UserFactory.php');
```

## lang\_path()

Функция `lang_path` возвращает полный путь к каталогу `lang` вашего приложения. Вы также можете использовать функцию `lang_path` для генерации полного пути к указанному файлу внутри этого каталога:

```
$path = lang_path();  
  
$path = lang_path('en/messages.php');
```

По умолчанию в структуре приложения Laravel отсутствует каталог `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды Artisan `lang:publish`.

## mix()

Функция `mix` возвращает путь к версионированному файлу Mix:

```
$path = mix('css/app.css');
```

## public\_path()

Функция `public_path` возвращает полный путь к каталогу `public` вашего приложения. Вы также можете использовать функцию `public_path` для генерации полного пути к заданному файлу в публичном каталоге:

```
$path = public_path();  
  
$path = public_path('css/app.css');
```

## resource\_path()

Функция `resource_path` возвращает полный путь к каталогу `resources` вашего приложения. Вы также можете использовать функцию `resource_path`, чтобы сгенерировать полный путь к заданному файлу в каталоге исходников:

```
$path = resource_path();  
  
$path = resource_path('sass/app.scss');
```

## storage\_path()

Функция `storage_path` возвращает полный путь к каталогу `storage` вашего приложения. Вы также можете использовать функцию `storage_path` для генерации полного пути к заданному файлу в каталоге хранилища:

```
$path = storage_path();  
  
$path = storage_path('app/file.txt');
```

## # URL-адреса

### action()

Функция `action` генерирует URL-адрес для переданного действия контроллера:

```
use App\Http\Controllers\HomeController;  
  
$url = action([HomeController::class, 'index']);
```

Если метод принимает параметры маршрута, вы можете передать их как второй аргумент методу:

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

### asset()

Функция `asset` генерирует URL для исходника (прим. перев.: директория `resources`), используя текущую схему запроса (HTTP или HTTPS):

```
$url = asset('img/photo.jpg');
```

Вы можете настроить хост URL исходников, установив переменную `ASSET_URL` в вашем файле `.env`. Это может быть полезно, если вы размещаете свои исходники на внешнем сервисе, таком как Amazon S3 или другой CDN:

```
// ASSET_URL=http://example.com/assets  
  
$url = asset('img/photo.jpg'); // http://example.com/assets/img/photo.jpg
```

## route()

Функция `route` генерирует URL для переданного [именованного маршрута](#):

```
$url = route('route.name');
```

Если маршрут принимает параметры, вы можете передать их в качестве второго аргумента методу:

```
$url = route('route.name', ['id' => 1]);
```

По умолчанию функция `route` генерирует абсолютный URL. Если вы хотите создать относительный URL, вы можете передать `false` в качестве третьего аргумента:

```
$url = route('route.name', ['id' => 1], false);
```

## secure\_asset()

Функция `secure_asset` генерирует URL для исходника, используя HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

## secure\_url()

Функция `secure_url` генерирует полный URL-адрес для указанного пути, используя HTTPS. Дополнительные сегменты URL могут быть переданы во втором аргументе функции:

```
$url = secure_url('user/profile');

$url = secure_url('user/profile', [1]);
```

## to\_route()

Функция `to_route` генерирует [HTTP-ответ перенаправления](#) для заданного [именованного маршрута](#):

```
return to_route('users.show', ['user' => 1]);

return to_route('users.show', ['user' => 1], 302, ['X-Framework' => 'Laravel']);
```

При необходимости вы можете передать методу `to_route` код состояния HTTP, который должен быть присвоен перенаправлению, а также любые дополнительные заголовки ответа в качестве третьего и четвёртого аргументов:

## url()

Функция `url` генерирует полный URL-адрес для указанного пути:

```
$url = url('user/profile');

$url = url('user/profile', [1]);
```

Если путь не указан, будет возвращен экземпляр `Illuminate\Routing\UrlGenerator`:

```
$current = url()->current();

$full = url()->full();

$previous = url()->previous();
```

## # Разное

### abort()

Функция `abort` генерирует [HTTP-исключение](#), которое будет обработано [обработчиком исключения](#):

```
abort(403);
```

Вы также можете указать текст ответа исключения и пользовательские заголовки ответа, которые должны быть отправлены в браузер:

```
abort(403, 'Unauthorized.', $headers);
```

### abort\_if()

Функция `abort_if` генерирует исключение HTTP, если переданное логическое выражение имеет значение `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

Подобно методу `abort`, вы также можете указать текст ответа исключения третьим аргументом и массив пользовательских заголовков ответа в качестве четвертого аргумента.

### abort\_unless()

Функция `abort_unless` генерирует исключение HTTP, если переданное логическое выражение оценивается как `false`:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

Подобно методу `abort`, вы также можете указать текст ответа исключения третьим аргументом и массив пользовательских заголовков ответа в качестве четвертого аргумента.

## app()

Функция `app` возвращает экземпляр [контейнера служб](#):

```
$container = app();
```

Вы можете передать имя класса или интерфейса для извлечения его из контейнера:

```
$api = app('HelpSpot\API');
```

## auth()

Функция `auth` возвращает экземпляр [автентификатора](#). Вы можете использовать его вместо фасада `Auth` для удобства:

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

При необходимости вы можете указать, к какому экземпляру охранника вы хотите получить доступ:

```
$user = auth('admin')->user();
```

## back()

Функция `back` генерирует [HTTP-ответ перенаправления](#) в предыдущее расположение пользователя:

```
return back($status = 302, $headers = [], $fallback = '/');

return back();
```

## bcrypt()

Функция `bcrypt` [хеширует](#) переданное значение, используя Bcrypt. Вы можете использовать его как альтернативу фасаду `Hash`:

```
$password = bcrypt('my-secret-password');
```

## blank()

Функция `blank` проверяет, является ли переданное значение «пустым»:

```
blank('');
blank('   ');
blank(null);
blank(collect());  
  
// true  
  
blank(0);
blank(true);
blank(false);  
  
// false
```

Обратной функции `blank` является функция [filled](#).

## broadcast()

Функция `broadcast` [транслирует](#) переданное [событие](#) своим слушателям:

```
broadcast(new UserRegistered($user));  
  
broadcast(new UserRegistered($user))->toOthers();
```

## cache()

Функция `cache` используется для получения значений из [кеша](#). Если переданный ключ не существует в кеше, будет возвращено необязательное значение по умолчанию:

```
$value = cache('key');  
  
$value = cache('key', 'default');
```

Вы можете добавлять элементы в кеш, передавая массив пар ключ / значение в функцию. Вы также должны передать количество секунд или продолжительность актуальности кешированного значения:

```
cache(['key' => 'value'], 300);  
cache(['key' => 'value'], now()->addSeconds(10));
```

## class\_uses\_recursive()

Функция `class_uses_recursive` возвращает все трейты, используемые классом, включая трейты, используемые всеми его родительскими классами:

```
$traits = class_uses_recursive(App\Models\User::class);
```

## collect()

Функция `collect` создает экземпляр [коллекции](#) переданного значения:

```
$collection = collect(['taylor', 'abigail']);
```

## config()

Функция `config` получает значение переменной [конфигурации](#). Доступ к значениям конфигурации можно получить с помощью «точечной нотации», включающую имя файла и параметр, к которому вы хотите получить доступ. Значение по умолчанию может быть указано и возвращается, если опция конфигурации не существует:

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

Вы можете установить переменные конфигурации на время выполнения скрипта, передав массив пар ключ / значение. Однако обратите внимание, что эта функция влияет только на значение конфигурации для текущего запроса и не обновляет фактические значения конфигурации:

```
config(['app.debug' => true]);
```

## context()

Функция `context` получает значение из [текущего контекста](#). Может быть указано значение по умолчанию, которое возвращается, если ключ контекста не существует:

```
$value = context('trace_id');  
  
$value = context('trace_id', $default);
```

Вы можете установить значения контекста, передав массив пар ключ/значение:

```
use Illuminate\Support\Str;  
  
context(['trace_id' => Str::uuid()->toString()]);
```

## cookie()

Функция `cookie` создает новый экземпляр [Cookie](#):

```
$cookie = cookie('name', 'value', $minutes);
```

## csrf\_field()

Функция `csrf_field` генерирует HTML «скрытого» поля ввода, содержащее значение токена CSRF. Например, используя [синтаксис Blade](#):

```
{{ csrf_field() }}
```

## csrf\_token()

Функция `csrf_token` возвращает значение текущего токена CSRF:

```
$token = csrf_token();
```

## decrypt()

Функция `decrypt` расшифровывает предоставленное значение. Вы можете использовать эту функцию в качестве альтернативы фасаду `Crypt`.

```
$password = decrypt($value);
```

## dd()

Функция `dd` выводит переданные переменные и завершает выполнение скрипта:

```
dd($value);  
dd($value1, $value2, $value3, ...);
```

Если вы не хотите останавливать выполнение вашего скрипта, используйте вместо этого функцию `dump`.

## dispatch()

Функция `dispatch` помещает переданное задание в очередь заданий Laravel:

```
dispatch(new App\Jobs\SendEmails);
```

## dispatch\_sync()

Функция `dispatch_sync` помещает предоставленную задачу в очередь синхронно для немедленной обработки:

```
dispatch_sync(new App\Jobs\SendEmails);
```

## dump()

Функция `dump` выводит переданные переменные:

```
dump($value);  
dump($value1, $value2, $value3, ...);
```

Если вы хотите прекратить выполнение скрипта после вывода переменных, используйте вместо этого функцию `dd`.

## encrypt()

Функция `encrypt` шифрует предоставленное значение. Вы можете использовать эту функцию в качестве альтернативы фасаду `Crypt`.

```
$secret = encrypt('my-secret-value');
```

## env()

Функция `env` возвращает значение переменной окружения или значение по умолчанию:

```
$env = env('APP_ENV');  
  
$env = env('APP_ENV', 'production');
```

Если вы выполнили команду `config:cache` во время процесса развертывания, вы должны быть уверены, что вызываете функцию `env` только из файлов конфигурации. Как только конфигурации будут кешированы, файл `.env` не будет загружаться, и все вызовы функции `env` будут возвращать `null`.

## event()

Функция `event` отправляет переданное событие своим слушателям:

```
event(new UserRegistered($user));
```

## fake()

Функция `fake` получает экземпляр [Faker](#) из контейнера, что может быть полезно при создании фиктивных данных в фабриках моделей, наполнении базы данных, тестировании и создании макетов представлений:

```
@for($i = 0; $i < 10; $i++)
    <dl>
        <dt>Name</dt>
        <dd>{{ fake()->name() }}</dd>
        <dt>Email</dt>
        <dd>{{ fake()->unique()->safeEmail() }}</dd>
    </dl>
@endfor
```

По умолчанию функция `fake` будет использовать опцию `appfaker_locale` из файла конфигурации `config/app.php`. Обычно этот параметр конфигурации задается через переменную среды `APP_FAKER_LOCALE`. Вы также можете указать локализацию, передав ее в функцию `fake`. Для каждой локализации будет создан свой собственный экземпляр:

```
fake('nl_NL')->name()
```

## filled()

Функция `filled` проверяет, является ли переданное значение не «пустым»:

```
filled(0);
filled(true);
filled(false);

// true

filled('');
filled(' ');
filled(null);
filled(collect());

// false
```

Обратной функции `filled` является функция [blank](#).

## info()

Функция `info` запишет информацию в [журнал](#):

```
info('Some helpful information!');
```

Также функции может быть передан массив контекстных данных:

```
info('User login attempt failed.', ['id' => $user->id]);
```

## literal()

Функция `literal` создает новый экземпляр [stdClass](#) с заданными именованными аргументами в качестве свойств:

```
$obj = literal(
    name: 'Joe',
    languages: ['PHP', 'Ruby'],
);

$obj->name; // 'Joe'
$obj->languages; // ['PHP', 'Ruby']
```

## logger()

Функцию `logger` можно использовать для записи сообщения уровня `debug` в [журнал](#):

```
logger('Debug message');
```

Также функции может быть передан массив контекстных данных:

```
logger('User has logged in.', ['id' => $user->id]);
```

Если функции не передано значение, то будет возвращен экземпляр [регистратора](#):

```
logger()->error('You are not allowed here.');
```

## method\_field()

Функция `method_field` генерирует HTML «скрытого» поле ввода, содержащее поддельное значение HTTP-метода формы. Например, используя [синтаксис Blade](#):

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

## now()

Функция `now` создает новый экземпляр `Illuminate\Support\Carbon` для текущего времени:

```
$now = now();
```

## old()

Функция `old` [возвращает](#) значение [прежнего ввода](#), краткосрочно сохраненное в сессии:

```
$value = old('value');

$value = old('value', 'default');
```

Поскольку значение по умолчанию, предоставляемое вторым аргументом функции `old`, часто является атрибутом модели Eloquent, Laravel позволяет вам просто передать всю модель Eloquent в качестве второго аргумента функции `old`. При этом Laravel предполагает, что первый аргумент, предоставленный функции `old`, – это имя атрибута Eloquent, которое следует считать значением по умолчанию:

```
{{ old('name', $user->name) }}

// Is equivalent to...

{{ old('name', $user) }}
```

## once()

Функция `once` выполняет заданный обратный вызов и кэширует результат в памяти на время запроса. Любые последующие вызовы функции `once` с тем же обратным вызовом будут возвращать ранее кэшированный результат:

```
function random(): int
{
    return once(function () {
        return random_int(1, 1000);
    });
}

random(); // 123
random(); // 123 (cached result)
random(); // 123 (cached result)
```

Когда функция `once` выполняется из экземпляра объекта, кэшированный результат будет уникальным для этого экземпляра объекта:

```
<?php

class NumberService
{
    public function all(): array
    {
        return once(fn () => [1, 2, 3]);
    }
}

$service = new NumberService;

$service->all();
$service->all(); // (cached result)

$secondService = new NumberService;

$secondService->all();
$secondService->all(); // (cached result)
```

## optional()

Функция `optional` принимает любой аргумент и позволяет вам получать доступ к свойствам или вызывать методы этого объекта. Если переданный объект

имеет значение `null`, свойства и методы будут возвращать также `null` вместо вызова ошибки:

```
return optional($user->address)->street;  
{!! old('name', optional($user)->name) !!}
```

Функция `optional` также принимает замыкание в качестве второго аргумента. Замыкание будет вызвано, если значение, указанное в качестве первого аргумента, не равно `null`:

```
return optional(User::find($id), function (User $user) {  
    return $user->name;  
});
```

## policy()

Функция `policy` извлекает экземпляр [ПОЛИТИКИ](#) для переданного класса:

```
$policy = policy(App\Models\User::class);
```

## redirect()

Функция `redirect` возвращает [HTTP-ответ перенаправления](#) или возвращает экземпляр перенаправителя, если вызывается без аргументов:

```
return redirect($to = null, $status = 302, $headers = [], $https = null);  
  
return redirect('/home');  
  
return redirect()->route('route.name');
```

## report()

Функция `report` сообщит об исключении, используя ваш [обработчик исключений](#):

```
report($e);
```

Функция `report` также принимает строку в качестве аргумента. Когда в функцию передается строка, она создает исключение с переданной строкой в качестве сообщения:

```
report('Something went wrong.');
```

## report\_if()

Функция `report_if` будет сообщать об исключении с использованием вашего обработчика исключений, если заданное условие является `true`:

```
report_if($shouldReport, $e);  
  
report_if($shouldReport, 'Something went wrong.');
```

## report\_unless()

Функция `report_unless` будет сообщать об исключении с использованием вашего обработчика исключений, если заданное условие является `false`:

```
report_unless($reportingDisabled, $e);  
  
report_unless($reportingDisabled, 'Something went wrong.');
```

## request()

Функция `request` возвращает экземпляр текущего запроса или получает значение поля ввода из текущего запроса:

```
$request = request();  
  
$value = request('key', $default);
```

## rescue()

Функция `rescue` выполняет переданное замыкание и перехватывает любые исключения, возникающие во время его выполнения. Все перехваченные

исключения будут отправлены вашему [обработчику исключений](#); однако, обработка запроса будет продолжена:

```
return rescue(function () {
    return $this->method();
});
```

Вы также можете передать второй аргумент функции `rescue`. Этот аргумент будет значением «по умолчанию», которое должно быть возвращено, если во время выполнения замыкание возникнет исключение:

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

Функции `rescue` может быть предоставлен аргумент `report`, чтобы определить, следует ли сообщать об исключении через функцию `report`:

```
return rescue(function () {
    return $this->method();
}, report: function (Throwable $throwable) {
    return $throwable instanceof InvalidArgumentException;
});
```

## resolve()

Функция `resolve` извлекает экземпляр связанного с переданным классом или интерфейсом, используя [контейнер служб](#):

```
$api = resolve('HelpSpot\API');
```

## response()

Функция `response` создает экземпляр [ответа](#) или получает экземпляр фабрики ответов:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

## retry()

Функция `retry` пытается выполнить переданную функцию, пока не будет достигнут указанный лимит попыток. Если функция не выбросит исключение, то будет возвращено её значение. Если функция выбросит исключение, то будет автоматически повторена. Если максимальное количество попыток превышено, будет выброшено исключение

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms between attempts...
}, 100);
```

Если вы хотите вручную вычислить количество миллисекунд, которое должно пройти между попытками, вы можете передать функцию в качестве третьего аргумента функции `retry`:

```
use Exception;

return retry(5, function () {
    // ...
}, function (int $attempt, Exception $exception) {
    return $attempt * 100;
});
```

Для удобства вы можете передать функции `retry` в качестве первого аргумента массив. Этот массив будет использоваться для определения интервала в миллисекундах между последующими попытками:

```
return retry([100, 200], function () {
    // Sleep for 100ms on first retry, 200ms on second retry...
});
```

Чтобы повторить попытку только при определенных условиях, вы можете передать функцию, определяющее это условие, в качестве четвертого аргумента функции `retry`:

```
use Exception;

return retry(5, function () {
    // ...
}, 100, function ($exception) {
    return $exception instanceof RetryException;
});
```

## session()

Функция `session` используется для получения или задания значений [сессии](#):

```
$value = session('key');
```

Вы можете установить значения, передав массив пар ключ / значение в функцию:

```
session(['chairs' => 7, 'instruments' => 3]);
```

Если в функцию не передано значение, то будет возвращен экземпляр хранилища сессий:

```
$value = session()->get('key');

session()->put('key', $value);
```

## tap()

Функция `tap` принимает два аргумента: произвольное значение и замыкание.

Значение будет передано в замыкание, а затем возвращено функцией `tap`.

Возвращаемое значение замыкания не имеет значения:

```
$user = tap(User::first(), function (User $user) {
    $user->name = 'taylor';
```

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

Если замыкание не передано функции `tap`, то вы можете вызвать любой метод с указанным значением. Возвращаемое значение вызываемого метода всегда будет изначально указанное, независимо от того, что метод фактически возвращает в своем определении. Например, метод Eloquent `update` обычно возвращает целочисленное значение. Однако, мы можем заставить метод возвращать саму модель, увязав вызов метода `update` с помощью функции `tap`:

```
$user = tap($user)->update([  
    'name' => $name,  
    'email' => $email,  
]);
```

Чтобы добавить к своему классу метод `tap`, используйте трейт `Illuminate\Support\Traits\Tappable` в вашем классе. Метод `tap` этого трейта принимает замыкание в качестве единственного аргумента. Сам экземпляр объекта будет передан замыканию, а затем будет возвращен методом `tap`:

```
return $user->tap(function (User $user) {  
    //  
});
```

## throw\_if()

Функция `throw_if` выбрасывает переданное исключение, если указанное логическое выражение оценивается как `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);  
  
throw_if(  
    ! Auth::user()->isAdmin(),  
    AuthorizationException::class,  
    'You are not allowed to access this page.'  
);
```

## throw\_unless()

Функция `throw_unless` выбрасывает переданное исключение, если указанное логическое выражение оценивается как `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page.'
);
```

## today()

Функция `today` создает новый экземпляр `Illuminate\Support\Carbon` для текущей даты:

```
$today = today();
```

## trait\_uses\_recursive()

Функция `trait_uses_recursive` возвращает все трейты, используемые трейтом:

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

## transform()

Функция `transform` выполняет замыкание для переданного значения, если значение не пустое, и возвращает результат замыкания:

```
$callback = function (int $value) {
    return $value * 2;
};

$result = transform(5, $callback);

// 10
```

В качестве третьего параметра могут быть указаны значение по умолчанию или замыкание. Это значение будет возвращено, если переданное значение пустое:

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

## validator()

Функция `validator` создает новый экземпляр [валидатора](#) с указанными аргументами. Вы можете использовать его для удобства вместо фасада `Validator`:

```
$validator = validator($data, $rules, $messages);
```

## value()

Функция `value` возвращает переданное значение. Однако, если вы передадите замыкание в функцию, то замыкание будет выполнено, и будет возвращен его результат:

```
$result = value(true);

// true

$result = value(function () {
    return false;
});

// false
```

Функции `value` могут быть переданы дополнительные аргументы. Если первый аргумент является замыканием, то дополнительные параметры будут переданы в замыкание в качестве аргументов, в противном случае они будут проигнорированы:

```
$result = value(function (string $name) {
    return $name;
}, 'Taylor');

// 'Taylor'
```

## view()

Функция `view` возвращает экземпляр [представления](#):

```
return view('auth.login');
```

## with()

Функция `with` возвращает переданное значение. Если вы передадите замыкание в функцию в качестве второго аргумента, то замыкание будет выполнено и будет возвращен результат его выполнения:

```
$callback = function (mixed $value) {
    return is_numeric($value) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

## when()

Функция `when` возвращает заданное ей значение, если заданное условие имеет значение `true`. В противном случае возвращается `null`. Если замыкание передается в качестве второго аргумента функции, замыкание будет выполнено и будет возвращено его возвращаемое значение:

```
$value = when(true, 'Hello World');

$value = when(true, fn () => 'Hello World');
```

Функция `when` в первую очередь полезна для условного рендеринга атрибутов HTML:

```
<div {!! when($condition, 'wire:poll="calculate"' ) !!}>
...
</div>
```

## # Другие утилиты

### Benchmark

Иногда вам может потребоваться быстро оценить производительность определенных частей вашего приложения. В таких случаях вы можете воспользоваться классом `Benchmark` для измерения времени выполнения переданных обратных вызовов в миллисекундах:

```
<?php

use App\Models\User;
use Illuminate\Support\Benchmark;

Benchmark::dd(fn () => User::find(1)); // 0.1 ms

Benchmark::dd([
    'Scenario 1' => fn () => User::count(), // 0.5 ms
    'Scenario 2' => fn () => User::all()->count(), // 20.0 ms
]);
```

По умолчанию переданные обратные вызовы будут выполнены один раз (одна итерация), и их длительность будет отображена в браузере / консоли.

Чтобы выполнить обратный вызов более одного раза, вы можете указать количество итераций вторым аргументом метода. При выполнении обратного вызова более одного раза класс `Benchmark` вернет среднее количество миллисекунд, затраченных на выполнение обратного вызова за все итерации:

```
Benchmark::dd(fn () => User::count(), iterations: 10); // 0.5 ms
```

Иногда вам может потребоваться измерить время выполнения обратного вызова, сохраняя при этом значение, возвращаемое обратным вызовом. Метод `value` вернет кортеж, содержащий значение, возвращаемое обратным вызовом, и количество миллисекунд, затраченных на выполнение обратного вызова:

```
[$count, $duration] = Benchmark::value(fn () => User::count());
```

## Даты

Laravel включает в себя [Carbon](#), мощную библиотеку для манипулирования датой и временем. Чтобы создать новый экземпляр [Carbon](#), вы можете вызвать функцию `now`. Эта функция доступна глобально в вашем приложении Laravel:

```
$now = now();
```

Или же вы можете создать новый экземпляр [Carbon](#), используя класс [Illuminate\Support\Carbon](#):

```
use Illuminate\Support\Carbon;  
  
$now = Carbon::now();
```

Подробное описание [Carbon](#) и его функций можно найти в [официальной документации Carbon](#).

## Отложенные функции

Отложенные функции в настоящее время находятся на стадии бета-тестирования, пока мы собираем отзывы сообщества.

Хотя [задания в очереди](#) Laravel позволяют ставить задачи в очередь для фоновой обработки, иногда у вас могут возникнуть простые задачи, которые вы хотели бы отложить без настройки или обслуживания долго работающего обработчика очереди.

Отложенные функции позволяют отложить выполнение закрытия до тех пор, пока HTTP-ответ не будет отправлен пользователю, что позволяет вашему приложению

чувствовать себя быстрым и отзывчивым. Чтобы отложить выполнение замыкания, просто передайте его функции `Illuminate\Support\defer`:

```
use App\Services\Metrics;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use function Illuminate\Support\defer;

Route::post('/orders', function (Request $request) {
    // Create order...

    defer(fn () => Metrics::reportOrder($order));

    return $order;
});
```

По умолчанию отложенные функции будут выполняться только в том случае, если HTTP-ответ, команда Artisan или задание в очереди, из которого вызывается `Illuminate\Support\defer`, завершаются успешно. Это означает, что отложенные функции не будут выполняться, если запрос приведет к HTTP-ответу `4xx` или `5xx`. Если вы хотите, чтобы отложенная функция выполнялась всегда, вы можете связать метод `always` с вашей отложенной функцией:

```
defer(fn () => Metrics::reportOrder($order))->always();
```

## Отмена отложенных функций

Если вам нужно отменить отложенную функцию до ее выполнения, вы можете использовать метод `forget`, чтобы отменить функцию по ее имени. Чтобы назвать отложенную функцию, укажите второй аргумент функции `Illuminate\Support\defer`:

```
defer(fn () => Metrics::report(), 'reportMetrics');

defer()->forget('reportMetrics');
```

## Совместимость отложенных функций

Если вы обновились до Laravel 11.x из приложения Laravel 10.x и скелет вашего приложения все еще содержит файл `app/Http/Kernel.php`, вам следует добавить

промежуточное программное обеспечение `InvokeDeferredCallbacks` в начало свойства `$middleware` ядра:

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\InvokeDeferredCallbacks::class, // [tl! a
    \App\Http\Middleware\TrustProxies::class,
    // ...
];
```

## Лотерея

Класс лотереи Laravel может использоваться для выполнения обратных вызовов на основе заданных шансов. Это может быть особенно полезно, когда вы хотите выполнить код только для определенного процента ваших входящих запросов:

```
use Illuminate\Support\Lottery;

Lottery::odds(1, 20)
    ->winner(fn () => $user->won())
    ->loser(fn () => $user->lost())
    ->choose();
```

Вы можете комбинировать класс лотереи Laravel с другими функциями Laravel. Например, вы можете захотеть сообщать обработчику исключений только о небольшом проценте медленных запросов. А поскольку класс лотереи является вызываемым, мы можем передать экземпляр класса в любой метод, который принимает вызываемые объекты:

```
use Carbon\CarbonInterval;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Lottery;

DB::whenQueryingForLongerThan(
    CarbonInterval::seconds(2),
    Lottery::odds(1, 100)->winner(fn () => report('Querying > 2 seconds.'))
);
```

## Тестирование лотерей

Laravel предоставляет несколько простых методов, которые позволяют легко тестируировать вызовы лотереи в вашем приложении:

```
// Лотерея всегда выигравшая...
Lottery::alwaysWin();

// Лотерея всегда проигравшая...
Lottery::alwaysLose();

// Выигрыш, проигрыш, затем вернуться к нормальному поведению...
Lottery::fix([true, false]);

// Вернуться к нормальному поведению...
Lottery::determineResultsNormally();
```

## Pipeline

Фасад [Pipeline](#) в Laravel предоставляет удобный способ “прокидывания” ввода через серию вызовов классов, замыканий или вызываемых объектов, предоставляя каждому классу возможность проверить или изменить входные данные и вызвать следующий элемент в цепочке вызовов пайплайна:

```
use Closure;
use App\Models\User;
use Illuminate\Support\Facades\Pipeline;

$user = Pipeline::send($user)
    ->through([
        function (User $user, Closure $next) {
            // ...

            return $next($user);
        },
        function (User $user, Closure $next) {
            // ...

            return $next($user);
        },
    ])
    ->then(fn (User $user) => $user);
```

Как видите, каждый вызываемый класс или замыкание указанное в pipeline получает входные данные и замыкание `$next`. Вызов замыкания `$next` приведет к

вызову следующего вызываемого объекта в пайплайне. Как вы могли заметить, это очень похоже на [middleware](#).

Когда последний вызываемый объект в пайплайне вызывает `$next`, будет выполнен объект, предоставленный методу `then`. Обычно этот вызываемый объект просто возвращает предоставленные входные данные.

Как было описано ранее, вы не ограничены предоставлением только замыканий в свой пайплайн. Вы также можете использовать вызываемые классы. Если предоставлено имя класса, экземпляр класса будет создан с использованием [контейнера служб Laravel](#), что позволяет внедрять зависимости в вызываемый класс:

```
$user = Pipeline::send($user)
    ->through([
        GenerateProfilePhoto::class,
        ActivateSubscription::class,
        SendWelcomeEmail::class,
    ])
    ->then(fn (User $user) => $user);
```

## Sleep

Класс `Sleep` в Laravel представляет собой легковесную обертку вокруг нативных функций PHP `sleep` и `usleep`, предоставляя большую тестируемость и удобный API для работы с временем:

```
use Illuminate\Support\Sleep;

$waiting = true;

while ($waiting) {
    Sleep::for(1)->second();

    $waiting = /* ... */;
}
```

Класс `Sleep` предоставляет разнообразные методы, позволяющие вам работать с различными единицами времени:

```

// Вернуть значение после сна...
$result = Sleep::for(1)->second()->then(fn () => 1 + 1);

// Спать, пока заданное значение истинно...
Sleep::for(1)->second()->while(fn () => shouldKeepSleeping());

// Приостановите выполнение на 90 секунд...
Sleep::for(1.5)->minutes();

// Приостановите выполнение на 2 секунды...
Sleep::for(2)->seconds();

// Pause execution for 500 milliseconds...
Sleep::for(500)->milliseconds();

// Приостановите выполнение на 500 миллисекунд...
Sleep::for(5000)->microseconds();

// Приостановить выполнение до заданного времени...
Sleep::until(now()->addMinute());

// Псевдоним функции PHP "sleep"...
Sleep::sleep(2);

// Псевдоним функции PHP "usleep"
Sleep::usleep(5000);

```

Чтобы легко объединять единицы времени, вы можете использовать метод `and`:

```
Sleep::for(1)->second()->and(10)->milliseconds();
```

## Тестирование Sleep

При тестировании кода, использующего класс `Sleep` или функции PHP `sleep`, выполнение вашего теста будет приостановлено. Как можно ожидать, это делает ваш пакет тестов значительно медленнее. Например, представьте, что вы тестируете следующий код:

```

$waiting = /* ... */;

$seconds = 1;

while ($waiting) {
    Sleep::for($seconds++)->seconds();
}

```

```
$waiting = /* ... */;  
}
```

Обычно тестирование этого кода займет как минимум одну секунду. К счастью, класс `Sleep` позволяет нам “подделывать” задержку, чтобы наш тестовый набор оставался быстрым:

Pest      PHPUnit

```
it('waits until ready', function () {  
    Sleep::fake();  
  
    // ...  
});
```

При подделке класса `Sleep` реальная задержка выполнения обходится, что приводит к более быстрому тестированию.

Как только класс `Sleep` был подделан, можно делать утверждения относительно ожидаемых “пауз”. Для иллюстрации давайте представим, что мы тестируем код, который приостанавливает выполнение три раза, при этом каждая задержка увеличивается на одну секунду. Используя метод `assertSequence`, мы можем проверить, что наш код “спал” нужное количество времени, сохраняя при этом скорость выполнения теста:

Pest      PHPUnit

```
it('checks if ready three times', function () {  
    Sleep::fake();  
  
    // ...  
  
    Sleep::assertSequence([  
        Sleep::for(1)->second(),  
        Sleep::for(2)->seconds(),  
        Sleep::for(3)->seconds(),  
    ]);  
})
```

Конечно же, класс `Sleep` предоставляет и другие утверждения, которые вы можете использовать при тестировании:

```
use Carbon\CarbonInterval as Duration;
use Illuminate\Support\Sleep;

// Утверждение, что sleep вызывали 3 раза...
Sleep::assertSleptTimes(3);

// Утверждение, что продолжительность сна...
Sleep::assertSlept(function (Duration $duration): bool {
    return /* ... */;
}, times: 1);

// Утверждение, что класс Sleep никогда не вызывался...
Sleep::assertNeverSlept();

// Утверждение, что, даже если был вызван Sleep, пауза в выполнении не наступила...
Sleep::assertInsomniac();
```



Иногда бывает полезно выполнять действие при каждом имитированном ожидании в коде вашего приложения. Для этого вы можете предоставить обратный вызов методу `whenFakingSleep`. В следующем примере мы используем помощники Laravel по [манипулированию временем](#), чтобы мгновенно продвинуть время на продолжительность каждого ожидания:

```
use Carbon\CarbonInterval as Duration;

$this->freezeTime();

Sleep::fake();

Sleep::whenFakingSleep(function (Duration $duration) {
    // Progress time when faking sleep...
    $this->travel($duration->totalMilliseconds()->milliseconds());
});
```

Поскольку прогрессирование времени является общим требованием, метод `fake` принимает аргумент `syncWithCarbon`, чтобы синхронизировать Carbon во время сна в тесте:

```
Sleep::fake(syncWithCarbon: true);

$start = now();
```

```
Sleep::for(1)->second();  
  
$start->diffForHumans(); // 1 second ago
```

Класс `Sleep` используется внутри Laravel при приостановке выполнения. Например, помощник `retry` использует класс `Sleep` при задержке, что обеспечивает лучшую тестируемость при использовании данного помощника.

# HTTP-клиент

# Введение

# Выполнение запросов

# Данные запроса

# Заголовки

# Аутентификация

# Время ожидания

# Повторные попытки

# Обработка ошибок

# Guzzle Middleware

# Параметры Guzzle

# Параллельные запросы

# Макросы

# Тестирование

# Фиктивные ответы

# Предотвращение случайных запросов

# Инспектирование запросов

# События

## # Введение

Laravel предлагает минимальный и выразительный API для HTTP-клиента [Guzzle](#), позволяющий быстро выполнять исходящие запросы для взаимодействия с другими веб-приложениями. Обертка вокруг Guzzle ориентирована на наиболее распространенные варианты использования и дает прекрасные возможности для разработчиков.

## # Выполнение запросов

Для отправки запросов вы можете использовать методы `head`, `get`, `post`, `put`, `patch` и `delete` фасада `Http`. Сначала давайте рассмотрим, как сделать основной запрос `GET`:

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

Метод `get` возвращает экземпляр `Illuminate\Http\Client\Response`, содержащий методы, которые можно использовать для получения информации об ответе:

```
$response->body() : string;
$response->json($key = null, $default = null) : mixed;
$response->object() : object;
$response->collect($key = null) : Illuminate\Support\Collection;
$response->resource() : resource;
$response->status() : int;
$response->ok() : bool;
$response->successful() : bool;
$response->redirect(): bool;
$response->failed() : bool;
$response->serverError() : bool;
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

Объект `Illuminate\Http\Client\Response` также реализует интерфейс `ArrayAccess` PHP, позволяющий напрямую получать доступ к данным ответа JSON:

```
return Http::get('http://example.com/users/1')['name'];
```

В дополнение к методам ответа, перечисленным выше, для определения того, имеет ли ответ заданный код состояния, можно использовать следующие методы:

```
$response->ok() : bool; // 200 OK
$response->created() : bool; // 201 Created
$response->accepted() : bool; // 202 Accepted
$response->noContent() : bool; // 204 No Content
$response->movedPermanently() : bool; // 301 Moved Permanently
$response->found() : bool; // 302 Found
$response->badRequest() : bool; // 400 Bad Request
$response->unauthorized() : bool; // 401 Unauthorized
```

```
$response->paymentRequired() : bool;           // 402 Payment Required
$response->forbidden() : bool;                  // 403 Forbidden
$response->notFound() : bool;                   // 404 Not Found
$response->requestTimeout() : bool;             // 408 Request Timeout
$response->conflict() : bool;                   // 409 Conflict
$response->unprocessableEntity() : bool;        // 422 Unprocessable Entity
$response->tooManyRequests() : bool;            // 429 Too Many Requests
$response->serverError() : bool;                // 500 Internal Server Error
```

## Шаблоны URI

HTTP-клиент также позволяет вам формировать URL-запросы с использованием [спецификации шаблонов URI](#). Для определения параметров URL, которые могут быть расширены в вашем шаблоне URI, вы можете использовать метод `withUrlParameters`:

```
Http::withUrlParameters([
    'endpoint' => 'https://laravel.com',
    'page' => 'docs',
    'version' => '11.x',
    'topic' => 'validation',
])->get('{+endpoint}/{page}/{version}/{topic}');
```

## Вывод информации о запросах

Если вы хотите получить информацию о сформированном экземпляре исходящего запроса перед его отправкой и прекратить выполнение скрипта, вы можете добавить метод `dd` в начало определения вашего запроса:

```
return Http::dd()->get('http://example.com');
```

## Данные запроса

При выполнении запросов `POST`, `PUT` и `PATCH` обычно отправляются дополнительные данные, поэтому эти методы принимают массив данных в качестве второго аргумента. По умолчанию данные будут отправляться с использованием типа содержимого `application/json`:

```
use Illuminate\Support\Facades\Http;
```

```
$response = Http::post('http://example.com/users', [
    'name' => 'Steve',
    'role' => 'Network Administrator',
]);
```

## Параметры GET-запроса

При выполнении запросов `GET` вы можете либо напрямую добавить строку запроса к URL, либо передать массив пар ключ / значение в качестве второго аргумента метода `get`:

```
$response = Http::get('http://example.com/users', [
    'name' => 'Taylor',
    'page' => 1,
]);
```

В качестве альтернативы можно использовать метод `withQueryParameters`:

```
Http::retry(3, 100)->withQueryParameters([
    'name' => 'Taylor',
    'page' => 1,
])->get('http://example.com/users')
```

## Отправка запросов с передачей данных в URL-кодированной строке

Если вы хотите отправлять данные с использованием типа содержимого `application/x-www-form-urlencoded`, то вы должны вызвать метод `asForm` перед выполнением запроса:

```
$response = Http::asForm()->post('http://example.com/users', [
    'name' => 'Sara',
    'role' => 'Privacy Consultant',
]);
```

## Отправка необработанного тела запроса

Вы можете использовать метод `withBody`, если хотите передать необработанное тело запроса при его выполнении. Тип контента может быть указан вторым

аргументом метода:

```
$response = Http::withBody(
    base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

## Составные запросы

Если вы хотите отправлять файлы в запросах, состоящих из нескольких частей, необходимо вызвать метод `attach` перед выполнением запроса. Этот метод принимает имя файла и его содержимое. При желании вы можете указать третий аргумент, который будет считаться именем файла, в то время как четвертый аргумент может быть использован для предоставления заголовков, связанных с файлом:

```
$response = Http::attach(
    'attachment', file_get_contents('photo.jpg'), 'photo.jpg', [
        'Content-Type' => 'image/jpeg'
    ]
)->post('http://example.com/attachments');
```

Вы также можете передать потоковый ресурс вместо передачи необработанного содержимого файла:

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
    'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

## Заголовки

Заголовки могут быть добавлены к запросам с помощью метода `withHeaders`. Метод `withHeaders` принимает массив пар ключ / значение:

```
$response = Http::withHeaders([
    'X-First' => 'foo',
    'X-Second' => 'bar'
])->post('http://example.com/users', [
```

```
'name' => 'Taylor',  
]);
```

Вы можете использовать метод `accept`, чтобы указать тип контента, который ваше приложение ожидает в ответ на ваш запрос:

```
$response = Http::accept('application/json')->get('http://example.com/users');
```

Для удобства вы можете использовать метод `acceptJson`, чтобы быстро указать, что ваше приложение ожидает тип содержимого `application/json` в ответ на ваш запрос:

```
$response = Http::acceptJson()->get('http://example.com/users');
```

Метод `withHeaders` объединяет новые заголовки с существующими заголовками запроса. При необходимости вы можете полностью заменить все заголовки, используя метод `replaceHeaders`:

```
$response = Http::withHeaders([  
    'X-Original' => 'foo',  
)->replaceHeaders([  
    'X-Replacement' => 'bar',  
)->post('http://example.com/users', [  
    'name' => 'Taylor',  
]);
```

## Аутентификация

Вы можете указать данные **basic** и **digest** аутентификации, используя методы `withBasicAuth` и `withDigestAuth`, соответственно:

```
// Basic HTTP-аутентификация ...  
$response = Http::withBasicAuth('taylor@laravel.com', 'secret')->post(/* ... */);  
  
// Digest HTTP-аутентификация ...  
$response = Http::withDigestAuth('taylor@laravel.com', 'secret')->post(/* ... */);
```

## Токены Bearer

Если вы хотите добавить токен в заголовок `Authorization` запроса, то используйте метод `withToken`:

```
$response = Http::withToken('token')->post(/* ... */);
```

## Время ожидания

Метод `timeout` используется для указания максимального количества секунд ожидания ответа. По умолчанию время ожидания HTTP-клиента истекает через 30 секунд::

```
$response = Http::timeout(3)->post(/* ... */);
```

Если указанный тайм-аут превышен, то будет выброшено исключение `Illuminate\Http\Client\ConnectionException`.

Вы можете указать максимальное количество секунд ожидания при попытке подключения к серверу, используя метод `connectTimeout`:

```
$response = Http::connectTimeout(3)->get(/* ... */);
```

## Повторные попытки

Если вы хотите, чтобы HTTP-клиент автоматически повторял запрос при возникновении ошибки клиента или сервера, то используйте метод `retry`. Метод `retry` принимает максимальное количество попыток выполнения запроса и количество миллисекунд, которые Laravel должен ждать между попытками:

```
$response = Http::retry(3, 100)->post(/* ... */);
```

Если вы хотите вручную вычислить количество миллисекунд, которые необходимо отвести на ожидание между попытками, вы можете передать замыкание в качестве второго аргумента методу `retry`:

```
use Exception;
```

```
$response = Http::retry(3, function (int $attempt, Exception $exception) {
    return $attempt * 100;
})->post(/* ... */);
```

Для удобства вы также можете указать массив в качестве первого аргумента метода `retry`. Этот массив будет использоваться для определения того, сколько миллисекунд нужно ожидать между последующими попытками:

```
$response = Http::retry([100, 200])->post(/* ... */);
```

При необходимости вы можете передать третий аргумент методу `retry`. Третий аргумент должен быть callable-функцией, которая определяет, следует ли на самом деле попытаться повторить попытку. Например, вы можете захотеть повторить запрос только в том случае, если начальный запрос обнаруживает исключение `ConnectionException`:

```
use Exception;
use Illuminate\Http\Client.PendingRequest;

$response = Http::retry(3, 100, function (Exception $exception, PendingRequest $request) {
    return $exception instanceof ConnectionException;
})->post(/* ... */);
```



Если попытка запроса завершится неудачей, вы можете захотеть внести изменение в запрос перед новой попыткой. Это можно сделать, изменив аргумент запроса, предоставленный вашему вызываемому объекту метода `retry`. Например, вы можете попробовать запрос с новым токеном авторизации, если первая попытка завершилась ошибкой аутентификации:

```
use Exception;
use Illuminate\Http\Client.PendingRequest;
use Illuminate\Http\Client\RequestException;

$response = Http::withToken($this->getToken())->retry(2, 0, function (Exception $exception) {
    if (! $exception instanceof RequestException || $exception->response->status() != 401)
        return false;
}

$request->withToken($this->getNewToken());
```

```
        return true;  
    })->post(/* ... */);
```

Если все запросы окажутся неуспешными, то будет выброшено исключение `Illuminate\Http\Client\RequestException`. Если вы хотите отключить это поведение, вы можете предоставить аргумент `throw` со значением `false`. При отключении, после всех попыток повтора, будет возвращен последний полученный клиентом ответ:

```
$response = Http::retry(3, 100, throw: false)->post(/* ... */);
```

Если все запросы завершаются неудачей из-за проблем с подключением, исключение `Illuminate\Http\Client\ConnectionException` все равно будет сгенерировано, даже если аргумент `throw` установлен в `false`.

## Обработка ошибок

В отличие от поведения Guzzle по умолчанию, обертка HTTP-клиента Laravel не генерирует исключений при возникновении ошибок клиента или сервера (ответы `400` и `500`, соответственно). Вы можете определить, была ли возвращена одна из этих ошибок, используя методы `successful`, `clientError`, или `serverError`:

```
// Определить, имеет ли ответ код состояния >= 200 and < 300...  
$response->successful();
```

```
// Определить, имеет ли ответ код состояния >= 400...  
$response->failed();
```

```
// Определить, имеет ли ответ код состояния 400 ...  
$response->clientError();
```

```
// Определить, имеет ли ответ код состояния 500 ...  
$response->serverError();
```

```
// Немедленно выполнить данную функцию обратного вызова, если произошла ошибка клиента
$response->onError(callable $callback);
```

## Выброс исключений

Если у вас есть экземпляр ответа и вы хотите выбросить исключение `Illuminate\Http\Client\RequestException`, если код состояния ответа указывает на ошибку клиента или сервера, используйте методы `throw` или `throwIf`:

```
use Illuminate\Http\Client\Response;

$response = Http::post(/* ... */);

// Выбросить исключение, если произошла ошибка клиента или сервера ...
$response->throw();

// Выбросить исключение, если произошла ошибка и данное условие истинно...
$response->throwIf($condition);

// Выбросить исключение, если произошла ошибка и данное замыкание принимает значение
$response->throwIf(fn (Response $response) => true);

// Выбросить исключение, если произошла ошибка и заданное условие равно false...
$response->throwUnless($condition);

// Выбросить исключение, если произошла ошибка и данное замыкание принимает значение
$response->throwUnless(fn (Response $response) => false);

// Выбросить исключение, если ответ имеет определенный код состояния...
$response->throwIfStatus(403);

// Выбросить исключение, если только ответ не содержит определенного кода состояния.
$response->throwUnlessStatus(200);

return $response['user']['id'];
```

Экземпляр `Illuminate\Http\Client\RequestException` имеет свойство `$response`, которое позволит вам проверить возвращенный ответ.

Метод `throw` возвращает экземпляр ответа, если ошибки не произошло, что позволяет вам использовать цепочку вызовов после метода `throw`:

```
return Http::post(/* ... */)->throw()->json();
```

Если вы хотите выполнить некоторую дополнительную логику до того, как будет сгенерировано исключение, вы можете передать замыкание методу `throw`. Исключение будет сгенерировано автоматически после вызова замыкания, поэтому вам не нужно повторно генерировать исключение изнутри замыкания:

```
use Illuminate\Http\Client\Response;
use Illuminate\Http\Client\RequestException;

return Http::post(/* ... */)->throw(function (Response $response, RequestException $e
    // ...
})->json();
```

## Guzzle Middleware

Поскольку HTTP-клиент Laravel работает на основе Guzzle, вы можете воспользоваться [Guzzle Middleware](#) для изменения исходящего запроса или анализа входящего ответа. Для изменения исходящего запроса зарегистрируйте middleware Guzzle с помощью метода `withRequestMiddleware`:

```
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\RequestInterface;

$response = Http::withRequestMiddleware(
    function (RequestInterface $request) {
        return $request->withHeader('X-Example', 'Value');
    }
)->get('http://example.com');
```

Точно так же вы можете осмотреть входящий HTTP-ответ, зарегистрировав middleware с помощью метода `withResponseMiddleware`:

```
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\ResponseInterface;

$response = Http::withResponseMiddleware(
    function (ResponseInterface $response) {
        $header = $response->getHeader('X-Example');
```

```
// ...  
  
        return $response;  
    }  
)->get('http://example.com');
```

## Глобальное Middleware

Иногда вы можете захотеть зарегистрировать middleware, которое применяется ко всем исходящим запросам и входящим ответам. Для этого вы можете использовать методы `globalRequestMiddleware` и `globalResponseMiddleware`. Обычно эти методы следует вызывать в методе `boot` файла `AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Http;  
  
Http::globalRequestMiddleware(fn ($request) => $request->withHeader(  
    'User-Agent', 'Example Application/1.0'  
));  
  
Http::globalResponseMiddleware(fn ($response) => $response->withHeader(  
    'X-Finished-At', now()->toDateTimeString()  
));
```

## Параметры Guzzle

Вы можете указать дополнительные [параметры запроса Guzzle](#) для исходящего запроса, используя метод `withOptions`. Метод `withOptions` принимает массив пар ключ/значение:

```
$response = Http::withOptions([  
    'debug' => true,  
])->get('http://example.com/users');
```

## Глобальные параметры

Чтобы настроить параметры по умолчанию для каждого исходящего запроса, вы можете использовать метод `globalOptions`. Обычно этот метод следует вызывать из метода `boot AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Http;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Http::globalOptions([
        'allow_redirects' => false,
    ]);
}
```

## # Параллельные запросы

Иногда вы можете захотеть выполнить несколько HTTP-запросов одновременно. Другими словами, вы хотите, чтобы несколько запросов отправлялись одновременно вместо того, чтобы отправлять их последовательно. Это может привести к значительному повышению производительности при взаимодействии с медленными HTTP API.

Вы можете сделать это с помощью метода `pool`. Метод `pool` принимает функцию с аргументом `Illuminate\Http\Client\Pool`, при помощи которого вы можете добавлять запросы в пул запросов для отправки:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->get('http://localhost/first'),
    $pool->get('http://localhost/second'),
    $pool->get('http://localhost/third'),
]);

return $responses[0]->ok() &&
    $responses[1]->ok() &&
    $responses[2]->ok();
```

Как видите, к каждому экземпляру ответа можно получить доступ в том порядке, в котором он был добавлен в пул. При желании вы можете назвать запросы с помощью метода `as`, что позволит вам получить доступ к соответствующим ответам по имени:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->as('first')->get('http://localhost/first'),
    $pool->as('second')->get('http://localhost/second'),
    $pool->as('third')->get('http://localhost/third'),
]);

return $responses['first']->ok();
```

## Customizing Concurrent Requests

Метод `pool` не может быть объединен с другими методами HTTP-клиента, такими как `withHeaders` или `middleware`. Если вы хотите применить пользовательские заголовки или middleware к пулу запросов, вы должны настроить эти параметры для каждого запроса в пуле:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$headers = [
    'X-Example' => 'example',
];

$responses = Http::pool(fn (Pool $pool) => [
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
    $pool->withHeaders($headers)->get('http://laravel.test/test'),
]);
```

## # Макросы

HTTP-клиент Laravel позволяет вам определять «макросы», которые могут служить плавным, выразительным механизмом для настройки общих путей запросов и заголовков при взаимодействии со службами в вашем приложении. Для начала вы можете определить макрос в методе `boot` класса `App\Providers\AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Http;
```

```
/**  
 * Bootstrap any application services.  
 */  
  
public function boot(): void  
{  
    Http::macro('github', function () {  
        return Http::withHeaders([  
            'X-Example' => 'example',  
        ])->baseUrl('https://github.com');  
    });  
}
```

После того, как ваш макрос настроен, вы можете вызвать его из любого места вашего приложения, чтобы создать ожидающий запрос с указанной конфигурацией:

```
$response = Http::github()->get('/');
```

## # Тестирование

Многие службы Laravel содержат функционал, помогающий вам легко и выразительно писать тесты, и HTTP-клиент Laravel не является исключением. Метод `fake` фасада `Http` позволяет указать HTTP-клиенту возвращать заглушенные / фиктивные ответы при выполнении запросов.

### Фиктивные ответы

Например, чтобы дать указание HTTP-клиенту возвращать пустые ответы с кодом состояния `200` на каждый запрос, вы можете вызвать метод `fake` без аргументов:

```
use Illuminate\Support\Facades\Http;  
  
Http::fake();  
  
$response = Http::post(/* ... */);
```

### Фальсификация конкретных URL

В качестве альтернативы вы можете передать массив методу `fake`. Ключи массива должны представлять шаблоны URL, которые вы хотите подделать, и связанные с ними ответы. Допускается использование метасимвола подстановки `*`. Любые запросы к URL-адресам, которые не были сфальсифицированы, будут выполнены фактически. Вы можете использовать метод `response` фасада `Http` для создания заглушек / фиктивных ответов для этих адресов:

```
Http::fake([
    // Заглушка JSON ответа для адресов GitHub ...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

    // Заглушка строкового ответа для адресов Google ...
    'google.com/*' => Http::response('Hello World', 200, $headers),
]);

```

Если вы хотите указать шаблон резервного URL-адреса, который будет заглушать все не сопоставленные URL-адреса, то используйте символ `*`:

```
Http::fake([
    // Заглушка JSON ответа для адресов GitHub ...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, ['Headers']),

    // Заглушка строкового ответа для всех остальных адресов ...
    '*' => Http::response('Hello World', 200, ['Headers']),
]);

```

## Фальсификация серии ответов

По желанию можно указать, что один URL должен возвращать серию фиктивных ответов в определенном порядке. Вы можете сделать это, используя метод `Http::sequence` для составления ответов:

```
Http::fake([
    // Заглушка серии ответов для адресов GitHub ...
    'github.com/*' => Http::sequence()
        ->push('Hello World', 200)
        ->push(['foo' => 'bar'], 200)
        ->pushStatus(404),
]);

```

Когда все ответы в этой последовательности будут использованы, любые дальнейшие запросы приведут к выбросу исключения. Если вы хотите указать ответ по умолчанию, который должен возвращаться, когда последовательность пуста, то используйте метод `whenEmpty`:

```
Http::fake([
    // Заглушка серии ответов для адресов GitHub ...
    'github.com/*' => Http::sequence()
        ->push('Hello World', 200)
        ->push(['foo' => 'bar'], 200)
        ->whenEmpty(Http::response()),
]);

```

Если вы хотите подделать серию ответов без указания конкретного шаблона URL, который следует подделать, то используйте метод `Http::fakeSequence`:

```
Http::fakeSequence()
    ->push('Hello World', 200)
    ->whenEmpty(Http::response());
```

## Анонимные фальсификаторы

Если вам требуется более сложная логика для определения того, какие ответы возвращать для определенных адресов, то вы можете передать замыкание методу `fake`. Это замыкание получит экземпляр `Illuminate\Http\Client\Request` и должно вернуть экземпляр ответа. В замыкании вы можете выполнить любую логику, необходимую для определения типа ответа, который нужно вернуть:

```
use Illuminate\Http\Client\Request;

Http::fake(function (Request $request) {
    return Http::response('Hello World', 200);
});
```

## Предотвращение случайных запросов

Если вы хотите удостовериться, что все запросы, отправленные через HTTP-клиент в рамках вашего отдельного теста или всего тестового набора, были поддельными, вы можете вызвать метод `preventStrayRequests`. После вызова этого метода любые

запросы, которые не имеют соответствующего поддельного ответа, вызовут исключение, вместо того чтобы делать фактический HTTP-запрос:

```
use Illuminate\Support\Facades\Http;

Http::preventStrayRequests();

Http::fake([
    'github.com/*' => Http::response('ok'),
]);

// An "ok" response is returned...
Http::get('https://github.com/laravel/framework');

// An exception is thrown...
Http::get('https://laravel.com');
```

## Инспектирование запросов

При фальсификации ответов вы можете иногда захотеть проверить запросы, которые получает клиент, чтобы убедиться, что ваше приложение отправляет правильные данные или заголовки. Вы можете сделать это, вызвав метод `Http::assertSent` после вызова `Http::fake`.

Метод `assertSent` принимает замыкание, которому будет передан экземпляр `Illuminate\Http\Client\Request` и, которое должно возвращать значение логического типа, указывающее, соответствует ли запрос вашим ожиданиям. Для успешного прохождения теста должен быть отправлен хотя бы один запрос, соответствующий указанным ожиданиям:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
    'X-First' => 'foo',
])->post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
```

```
return $request->hasHeader('X-First', 'foo') &&
    $request->url() == 'http://example.com/users' &&
    $request['name'] == 'Taylor' &&
    $request['role'] == 'Developer';
});
```

При необходимости вы можете утверждать, что конкретный запрос не был отправлен с помощью метода `assertNotSent`:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

Вы можете использовать метод `assertSentCount`, чтобы определить, сколько запросов было отправлено во время теста:

```
Http::fake();

Http::assertSentCount(5);
```

Или используйте метод `assertNothingSent`, чтобы утверждать, что во время теста не было отправлено никаких запросов:

```
Http::fake();

Http::assertNothingSent();
```

## Запись Запросов/Ответов

Вы можете использовать метод `recorded` для сбора всех запросов и соответствующих ответов. Метод `recorded` возвращает коллекцию массивов,

которая содержит экземпляры `Illuminate\Http\Client\Request` и `Illuminate\Http\Client\Response`:

```
Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded();

[$request, $response] = $recorded[0];
```

Кроме того, метод `recorded` принимает замыкание, которое получит экземпляры `Illuminate\Http\Client\Request` и `Illuminate\Http\Client\Response` и может быть использовано для фильтрации пар запрос/ответ в соответствии с вашими ожиданиями:

```
use Illuminate\Http\Client\Request;
use Illuminate\Http\Client\Response;

Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded(function (Request $request, Response $response) {
    return $request->url() !== 'https://laravel.com' &&
        $response->successful();
});
```

## # События

Laravel запускает три события в процессе отправки HTTP-запросов. Событие `RequestSending` запускается до отправки запроса, а событие `ResponseReceived` запускается после получения ответа на данный запрос. Событие `ConnectionFailed` запускается, если на данный запрос ответ не получен.

События `RequestSending` И `ConnectionFailed` содержат общедоступное свойство `$request`, которое вы можете использовать для проверки экземпляра `Illuminate\Http\Client\Request`. Аналогично, событие `ResponseReceived` содержит свойство `$request`, а также свойство `$response`, которое можно использовать для проверки экземпляра `Illuminate\Http\Client\Response`. Вы можете создать [прослушиватели событий](#) для этих событий в вашем приложении:

```
use Illuminate\Http\Client\Events\RequestSending;

class LogRequest
{
    /**
     * Handle the given event.
     */
    public function handle(RequestSending $event): void
    {
        // $event->request ...
    }
}
```

# Локализация интерфейса

## # Введение

- # Публикация языковых файлов
- # Конфигурирование языка по умолчанию
- # Язык плюрализатора

## # Определение строк перевода

- # Использование коротких ключей
- # Использование строк перевода в качестве ключей

## # Получение строк перевода

- # Замена параметров в строках перевода
- # Плюрализация

## # Переопределение языковых файлов пакета

## # Введение

По умолчанию, структура приложения Laravel не включает в себя каталог `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды Artisan `lang:publish`.

Функционал локализации Laravel предоставляют удобный способ извлечения строк разных языков, что позволяет легко поддерживать мультиязычность интерфейса вашего приложения.

Laravel предлагает два способа управления строками перевода. Во-первых, языковые строки могут храниться в файлах в каталоге `lang`. В этом каталоге могут быть подкаталоги для каждого языка, поддерживаемого приложением. Это подход,

который Laravel использует для управления строками перевода собственного функционала, например сообщений об ошибках валидации:

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

Или строки перевода могут быть определены в файлах JSON, которые помещаются в каталог `lang`. При таком подходе каждый язык, поддерживаемый вашим приложением, будет иметь соответствующий файл JSON в этом каталоге. Этот подход рекомендуется для приложений с большим количеством переводимых строк:

```
/lang
  en.json
  es.json
```

Мы обсудим каждый подход по управлению строками перевода в этой документации.

## Публикация языковых файлов

По умолчанию, структура приложения Laravel не включает в себя каталог `lang`. Если вы хотите настроить языковые файлы Laravel или создать собственные, вы можете создать каталог `lang` с помощью команды Artisan `lang:publish`. Команда `lang:publish` создаст каталог `lang` в вашем приложении и опубликует набор языковых файлов, используемых Laravel по умолчанию:

```
php artisan lang:publish
```

## Конфигурирование языка по умолчанию

Язык вашего приложения по умолчанию определяется в файле конфигурации `config/app.php` в опции `locale`. Обычно этот параметр устанавливается через переменную окружения `APP_LOCALE`. Вы вольны изменить это значение в соответствии с потребностями вашего приложения.

Также вы можете настроить “резервный язык”, который будет использоваться в случае отсутствия перевода для определенной строки на языке по умолчанию. Аналогично языку по умолчанию, резервный язык также настраивается в файле конфигурации `config/app.php`, обычно с использованием переменной окружения `APP_FALLBACK_LOCALE`.

Вы можете изменить язык по умолчанию для одного HTTP-запроса во время выполнения, используя метод `setLocale` фасада `App`:

```
use Illuminate\Support\Facades\App;

Route::get('/greeting/{locale}', function (string $locale) {
    if (! in_array($locale, ['en', 'es', 'fr'])) {
        abort(400);
    }

    App::setLocale($locale);

    // ...
});
```

## Определение текущего языка

Вы можете использовать методы `currentLocale` и `isLocale` фасада `App`, чтобы определить текущий язык или проверить соответствие указанного языка:

```
use Illuminate\Support\Facades\App;

$locale = App::currentLocale();

if (App::isLocale('en')) {
    // ...
}
```

## Язык плюрализатора

Вы можете настроить “множественное число” Laravel, которое используется Eloquent и другими частями фреймворка для преобразования единственных строк во множественные строки, чтобы использовать язык отличный от английского. Это можно сделать, вызвав метод `useLanguage` внутри метода `boot` одного из

проводеров служб вашего приложения. В настоящее время поддерживаемые языки множественного числа: [french](#), [norwegian-bokmal](#), [portuguese](#), [spanish](#) и [turkish](#):

```
use Illuminate\Support\Pluralizer;

/**
 * Загрузка сервисов приложения.
 */
public function boot(): void
{
    Pluralizer::useLanguage('spanish');

    // ...
}
```

Если вы настраиваете язык множественного числа, вы должны явно определить [имена таблиц](#) ваших моделей Eloquent.

## # Определение строк перевода

### Использование коротких ключей

Обычно строки перевода хранятся в файлах в каталоге `lang`. В этом каталоге должен быть подкаталог для каждого языка, поддерживаемого вашим приложением. Это подход, который Laravel использует для управления строками перевода собственного функционала, например сообщений об ошибках валидации:

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

Все языковые файлы возвращают массив строк с ключами. Например:

```
<?php  
  
// lang/en/messages.php  
  
return [  
    'welcome' => 'Welcome to our application!',  
];
```

Для языков, отличающихся территориально, вы должны называть языковые каталоги в соответствии со стандартом ISO 15897. Например, для британского английского следует использовать «en\_GB», а не «en-gb».

## Использование строк перевода в качестве ключей

Для приложений с большим количеством переводимых строк определение каждой строки с помощью «короткого ключа» может сбивать с толку при обращении к ключам в ваших шаблонах, и постоянно изобретать ключи для каждой строки перевода, поддерживаемой вашим приложением, затруднительно.

По этой причине Laravel предлагает определение строк перевода с использованием переводимой строки в качестве ключа «по умолчанию». Файлы перевода, которые используют строки перевода в качестве ключей, хранятся как файлы JSON в каталоге `lang`. Например, если ваше приложение имеет испанский перевод, то вы должны создать файл `lang/es.json`:

```
{  
    "I love programming.": "Me encanta programar."  
}
```

## Конфликты ключей и имен файлов

Вы не должны определять строковые ключи перевода, которые конфликтуют с именами файлов перевода. Например, перевод `__('Action')` для языка `NL` при условии существования файла `nl/action.php` и отсутствии файла `nl.json` приведет к тому, что переводчик Laravel вернет полное содержимое всего файла `nl/action.php`.

## # Получение строк перевода

Вы можете получить строки перевода из ваших языковых файлов с помощью глобального помощника `_`. Если вы используете «короткие ключи» для определения ваших строк перевода, то вы должны передать файл, содержащий ключ, и сам ключ в функцию `_`, используя «точечную нотацию». Например, давайте извлечем строку перевода `welcome` из языкового файла `lang/en/messages.php`:

```
echo _('messages.welcome');
```

Если указанная строка перевода не существует, то функция `_` вернет ключ строки перевода. Итак, используя приведенный выше пример, функция `_` вернет `messages.welcome`, если строка перевода не существует.

Если вы используете свои [строки перевода в качестве ключей перевода](#), то вы должны передать перевод вашей строки по умолчанию в функцию `_`:

```
echo _('I love programming.');
```

Опять же, если строка перевода не существует, то функция `_` вернет ключ строки перевода, который ей был передан.

Если вы используете [шаблонизатор Blade](#), то вы можете использовать синтаксис `{{}}` для вывода строки перевода:

```
{{ _('messages.welcome') }}
```

## Замена параметров в строках перевода

При желании вы можете определить метку-заполнитель в строках перевода. Все заполнители имеют префикс `:`. Например, вы можете определить приветственное сообщение с именем-заполнителем:

```
'welcome' => 'Welcome, :name',
```

Чтобы заменить заполнители при получении строки перевода, вы можете передать массив для замены в качестве второго аргумента функции `_`:

```
echo _('messages.welcome', ['name' => 'dayle']);
```

Если все буквы заполнителя заглавные или заполнитель имеет только первую заглавную букву, то переведенное значение будет с соответствующим регистром:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

## Форматирование объектов при подстановке

Когда вы пытаетесь использовать объект в качестве заполнителя перевода, Laravel автоматически вызывает метод `__toString` этого объекта. Метод `__toString` является одним из встроенных “магических” методов в PHP. Однако бывают случаи, когда у вас нет контроля над методом `__toString` определенного класса, особенно если это класс сторонней библиотеки.

В таких ситуациях Laravel предоставляет возможность зарегистрировать пользовательский обработчик форматирования для конкретного типа объекта. Для этого используется метод `stringable` фасада `Lang`. Метод `stringable` принимает замыкание, которое должно указать тип объекта, для которого выполняется форматирование. Обычно вызов метода `stringable` выполняется в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Lang;
use Money\Money;

/**
 * Настройка служб приложения.
 */
public function boot(): void
{
    Lang::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

# Плюрализация

Плюрализация – сложная задача, поскольку разные языки имеют множество сложных правил плюрализации; однако Laravel может помочь вам переводить строки по-разному в зависимости от правил множественного числа, которые вы определяете. Используя мета-символ |, вы можете различать формы единственного и множественного числа строки:

```
'apples' => 'There is one apple|There are many apples',
```

Конечно, множественное число также поддерживается при использовании [строк перевода в качестве ключей](#):

```
{  
    "There is one apple|There are many apples": "Hay una manzana|Hay muchas manzanas"  
}
```

Вы даже можете создать более сложные правила множественного числа, которые определяют строки перевода для нескольких диапазонов значений:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

После определения строки перевода, которая имеет параметры множественного числа, вы можете использовать функцию [trans\\_choice](#) для извлечения строки соответствующую указанному «количеству». В этом примере, поскольку количество больше единицы, возвращается форма множественного числа строки перевода:

```
echo trans_choice('messages.apples', 10);
```

Вы также можете определить метку-заполнитель в строках множественного числа. Эти заполнители могут быть заменены передачей массива в качестве третьего аргумента функции [trans\\_choice](#):

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',
```

```
echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

Если вы хотите отобразить целочисленное значение, переданное в функцию `trans_choice`, то вы можете использовать встроенный заполнитель `:count`:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

## # Переопределение языковых файлов пакета

Некоторые пакеты могут содержать собственные языковые файлы. Вместо того чтобы изменять строки файлов пакета, вы можете переопределить их, поместив файлы в каталог `lang/vendor/{package}/{locale}`.

Так, например, если вам нужно переопределить строки перевода на английский в `messages.php` для пакета с именем `skyrim/hearthfire`, вы должны поместить языковой файл в каталог: `lang/vendor/hearthfire/en/messages.php`. В этом файле вы должны определять только те строки перевода, которые хотите переопределить. Любые строки перевода, которые вы не меняете, все равно будут загружены из исходных языковых файлов пакета.

# Отправка электронной почты

## # Введение

- # Конфигурирование
- # Требования к драйверу и транспорту
- # Конфигурация аварийного переключения
- # Конфигурация Round Robin

## # Генерация отправлений

### # Написание отправлений

- # Конфигурирование отправителя
- # Конфигурирование шаблона
- # Данные шаблона
- # Вложения
- # Встраиваемые вложения
- # Объекты, которые можно прикреплять
- # Заголовки
- # Настройка Symfony Message

## # Отправления с разметкой Markdown

- # Генерация отправлений с разметкой Markdown
- # Написание сообщений с разметкой Markdown
- # Изменение компонентов

## # Отправка почты

- # Очередь почты

## # Отображение отправлений

- # Предварительный просмотр отправлений в браузере

## # Локализация отправлений

- # Предпочитаемые пользователем локализации

## # Тестирование

- # Тестирование содержимого отправлений
- # Тестирование отправки почтовых сообщений

## # Почта и локальная разработка

# События

# Пользовательские транспорты

# Дополнительные транспорты Symfony

## # Введение

Отправка электронной почты не должна быть сложной. Laravel предлагает чистый и простой почтовый API на базе популярного компонента [Symfony Mailer](#). Laravel и Symfony Mailer обеспечены драйверами для отправки электронной почты через SMTP, Mailgun, Postmark, Amazon SES и [sendmail](#), что позволяет быстро начать отправку почты через локальный или облачный сервис по вашему выбору.

## Конфигурирование

Почтовые службы Laravel могут быть настроены через конфигурационный файл [config/mail.php](#) вашего приложения. Каждая почтовая программа, настроенная в этом файле, может иметь свою собственную уникальную конфигурацию и даже свой собственный уникальный «транспорт», что позволяет вашему приложению использовать различные почтовые службы для отправки определенных сообщений электронной почты. Например, ваше приложение может использовать Postmark для отправки транзакционных писем, а Amazon SES – для массовых рассылок.

В конфигурационном файле [config/mail.php](#) вы найдете массив [mailers](#). Этот массив содержит образец записи конфигурации для каждого из основных почтовых драйверов / транспортов, поддерживаемых Laravel, в то время как значение конфигурации [default](#) определяет, какая почтовая программа будет использоваться по умолчанию, когда ваше приложение должно отправить сообщение электронной почты.

## Требования к драйверу и транспорту

Драйверы на основе API, такие, как Mailgun, Postmark, Resend и MailerSend часто проще в использовании и быстрее, чем отправка почты через SMTP-серверы. По возможности мы рекомендуем использовать один из этих драйверов.

## Драйвер Mailgun

Для использования драйвера Mailgun установите пакет Symfony's Mailgun Mailer с помощью Composer:

```
composer require symfony/mailgun-mailer symfony/http-client
```

Затем установите опцию `default` в конфигурационном файле `config/mail.php` вашего приложения в значение `mailgun` и добавьте следующий массив конфигурации в ваш массив `mailers`:

```
'mailgun' => [
    'transport' => 'mailgun',
    // 'client' => [
    //     'timeout' => 5,
    // ],
],
```

После настройки почтовой программы вашего приложения по умолчанию добавьте следующие параметры в файл конфигурации `config/services.php`:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.mailgun.net'),
    'scheme' => 'https',
],
```

Если вы не используете [регион Mailgun](#) США, то вы можете определить конечную точку своего региона в конфигурации файла `services`:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
    'scheme' => 'https',
],
```

## Драйвер Postmark

Чтобы использовать драйвер [Postmark](#), установите пакет Symfony's Postmark Mailer через Composer:

```
composer require symfony/postmark-mailer symfony/http-client
```

Затем установите опцию `default` в конфигурационном файле `config/mail.php` вашего приложения в значение `postmark`. После настройки основного почтового отправителя вашего приложения, убедитесь, что конфигурационный файл `config/services.php` содержит следующие опции:

```
'postmark' => [
    'token' => env('POSTMARK_TOKEN'),
],
```

Если вы хотите указать поток сообщений Postmark, который должен использоваться данной почтовой программой, вы можете добавить параметр конфигурации `message_stream_id` в массив конфигурации почтовой программы. Этот массив конфигурации можно найти в файле конфигурации вашего приложения `config/mail.php`:

```
'postmark' => [
    'transport' => 'postmark',
    'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
    // 'client' => [
    //     'timeout' => 5,
    // ],
],
```

Таким образом, вы также можете настроить несколько почтовых программ Postmark с разными потоками сообщений.

## Драйвер Resend

Чтобы использовать драйвер [Resend](#), установите PHP SDK Resend через Composer:

```
composer require resend/resend-php
```

Затем установите для параметра `default` в файле конфигурации вашего приложения `config/mail.php` значение `resend`. После настройки почтовой программы вашего приложения по умолчанию убедитесь, что ваш файл конфигурации `config/services.php` содержит следующие параметры:

```
'resend' => [
    'key' => env('RESEND_KEY'),
],
```

## Драйвер SES

Чтобы использовать драйвер Amazon SES, сначала необходимо установить Amazon AWS SDK для PHP. Вы можете установить эту библиотеку через менеджер пакетов Composer:

```
composer require aws/aws-sdk-php
```

Затем установите для параметра `default` в вашем файле конфигурации `config/mail.php` значение `ses` и убедитесь, что конфигурационный файл `config/services.php` содержит следующие параметры:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

Чтобы использовать AWS [временные учетные данные](#) через токен сеанса, вы можете добавить ключ `token` в конфигурацию SES вашего приложения:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'token' => env('AWS_SESSION_TOKEN'),
],
```

Чтобы взаимодействовать с [функциями управления подпиской](#) SES, вы можете вернуть заголовок `X-Ses-List-Management-Options` в массиве, возвращаемом методом `headers` почтового сообщения:

```
/**  
 * Get the message headers.  
 */  
public function headers(): Headers  
{  
    return new Headers(  
        text: [  
            'X-Ses-List-Management-Options' => 'contactListName=MyContactList;topicName=MyTopic;  
        ],  
    );  
}
```



Если вы хотите определить [дополнительные параметры](#), которые Laravel должен передать методу `SendEmail` AWS SDK при отправке сообщения электронной почты, вы можете определить массив `options` в конфигурации `ses`:

```
'ses' => [  
    'key' => env('AWS_ACCESS_KEY_ID'),  
    'secret' => env('AWS_SECRET_ACCESS_KEY'),  
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),  
    'options' => [  
        'ConfigurationSetName' => 'MyConfigurationSet',  
        'EmailTags' => [  
            ['Name' => 'foo', 'Value' => 'bar'],  
        ],  
    ],  
,
```

## Драйвер MailerSend

[MailerSend](#), сервис для отправки транзакционных электронных писем и SMS-сообщений, поддерживает свой собственный драйвер для Laravel, основанный на их API. Пакет, содержащий этот драйвер, можно установить с помощью менеджера пакетов Composer:

```
composer require mailersend/laravel-driver
```

После установки пакета добавьте переменную окружения `MAILERSEND_API_KEY` в файл `.env` вашего приложения. Кроме того, переменная окружения `MAIL_MAILER` должна быть определена как `mailersend`:

```
MAIL_MAILER=mailersend
MAIL_FROM_ADDRESS=app@yourdomain.com
MAIL_FROM_NAME="Имя приложения"

MAILERSEND_API_KEY=ваш-ключ-api
```

Наконец, добавьте MailerSend в массив `mailers` в файле конфигурации вашего приложения `config/mail.php`:

```
'mailersend' => [
    'transport' => 'mailersend',
],
```

Для получения дополнительной информации о MailerSend, включая инструкции по использованию хостинга шаблонов, обратитесь к [документации по драйверу MailerSend](#).

## Конфигурация аварийного переключения

Иногда внешняя служба, которую вы настроили для отправки почты вашего приложения, может не работать. В этих случаях может быть полезно определить одну или несколько резервных конфигураций доставки почты, которые будут использоваться в случае, если ваш основной драйвер доставки не работает.

Для этого вы должны определить почтовую программу в файле конфигурации вашего приложения `mail`, который использует транспорт `failover`. Массив конфигурации для почтовой программы `failover` вашего приложения должен содержать массив `mailers`, который определяет очередность, в которой почтовые программы должны быть выбраны для доставки:

```
'mailers' => [
    'failover' => [
        'transport' => 'failover',
        'mailers' => [
            'postmark',
```

```
    'mailgun',
    'sendmail',
],
],
// ...
],
```

После того как ваш почтовый агент аварийного переключения был определен, вы должны установить его как почтовую программу по умолчанию, используемую вашим приложением, указав ее имя как значение конфигурационного ключа `default` в файле конфигурации вашего приложения `mail`:

```
'default' => env('MAIL_MAILER', 'failover'),
```

## Конфигурация Round Robin

Транспорт `roundrobin` позволяет распределить вашу почтовую нагрузку между несколькими почтовыми клиентами. Чтобы начать, определите почтовый клиент в файле конфигурации `mail` вашего приложения, который использует транспорт `roundrobin`. Массив конфигурации почтового клиента вашего приложения `roundrobin` должен содержать массив `mailers`, который указывает, какие настроенные почтовые клиенты должны быть использованы для доставки:

```
'mailers' => [
    'roundrobin' => [
        'transport' => 'roundrobin',
        'mailers' => [
            'ses',
            'postmark',
        ],
    ],
],
// ...
],
```

После того как ваш почтовый клиент `round robin` был определен, вы должны установить этот клиент почты в качестве клиента по умолчанию, используемого вашим приложением, указав его имя в качестве значения ключа `default` в конфигурационном файле `mail` вашего приложения:

```
'default' => env('MAIL_MAILER', 'roundrobin'),
```

Транспорт `round robin` выбирает случайный почтовый клиент из списка настроенных почтовых клиентов, а затем переключается на следующий доступный почтовый клиент для каждого последующего электронного письма. В отличие от транспорта `failover`, который помогает достичь [высокой доступности](#), транспорт `roundrobin` обеспечивает [балансировку нагрузки](#).

## # Генерация отправлений

При создании приложений Laravel каждый тип электронной почты, отправляемой вашим приложением, представляется экземпляром класса `Illuminate\Mail\Mailable`. Эти классы хранятся в каталоге `app/Mail`. Не беспокойтесь, если вы не видите этот каталог в своем приложении, поскольку он будет сгенерирован для вас, когда вы создадите свой первый почтовый класс с помощью команды `make:mail Artisan`:

```
php artisan make:mail OrderShipped
```

## # Написание отправлений

После того как вы создали класс для отправки электронной почты, откройте его, чтобы мы могли рассмотреть его содержание. Конфигурация класса для отправки электронной почты выполняется в нескольких методах, включая методы `envelope`, `content` и `attachments`.

Метод `envelope` возвращает объект `Illuminate\Mail\Mailables\Envelope`, который определяет тему сообщения и, иногда, получателей сообщения. Метод `content` возвращает объект `Illuminate\Mail\Mailables\Content`, который определяет [шаблон Blade](#), который будет использоваться для генерации содержания сообщения.

## Конфигурирование отправителя

### Использование метода `Envelope`

Давайте сначала рассмотрим настройку отправителя электронной почты, то есть того, от кого будет отправлено письмо. Существует два способа настройки

отправителя. Во-первых, вы можете указать адрес отправителя в методе `envelope` вашего сообщения:

```
use Illuminate\Mail\Mailables\Address;
use Illuminate\Mail\Mailables\Envelope;

/**
 * Get the message envelope.
 */
public function envelope(): Envelope
{
    return new Envelope(
        from: new Address('jeffrey@example.com', 'Jeffrey Way'),
        subject: 'Order Shipped',
    );
}
```

Если необходимо, вы также можете указать адрес для ответа указав `replyTo`:

```
return new Envelope(
    from: new Address('jeffrey@example.com', 'Jeffrey Way'),
    replyTo: [
        new Address('taylor@example.com', 'Taylor Otwell'),
    ],
    subject: 'Заказ отправлен',
);
```

## Использование глобального адреса `from`

Однако, если ваше приложение использует один и тот же адрес `from` для всех своих электронных писем, вызов метода `from` в каждом создаваемом вами классе рассылки может стать громоздким. Вместо этого вы можете указать глобальный адрес отправителя в файле конфигурации `config/mail.php`. Этот адрес будет использоваться, если в почтовом классе не указан другой адрес в методе `from`:

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
],
```

Кроме того, вы можете определить глобальный адрес `reply_to` в конфигурационном файле `config/mail.php`:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

## Конфигурирование шаблона

Внутри метода `content` почтового класса вы можете использовать метод `view`, чтобы указать, какой шаблон следует использовать при отображении содержимого электронного письма. Поскольку каждое электронное письмо для визуализации своего содержимого обычно использует [шаблон Blade](#), вы получаете всю мощь и удобство механизма шаблонов Blade при создании HTML-кода электронной почты:

В методе `content` класса для отправки электронной почты вы можете определить метод `view`, то есть, какой шаблон должен использоваться при отображении содержимого электронного письма. Поскольку каждое электронное письмо обычно использует [шаблон Blade](#) для отображения своего содержания, у вас есть вся мощь и удобство шаблонизатора Blade при создании HTML-содержимого письма:

```
/*
 * Get the message content definition.
 *
 * @return $this
 */
public function content(): Content
{
    return new Content(
        view: 'mail.orders.shipped',
    );
}
```

Вы можете создать каталог `resources/views/emails` для размещения всех ваших шаблонов электронной почты; однако, вы можете размещать их где угодно в каталоге `resources/views`.

## Письма с обычным текстом

Если вы хотите указать версию письма для plain-text (обычный текст), вы можете указать его шаблон при определении `Content`. Как и параметр `view`, параметр `text` должен содержать имя шаблона, который будет использоваться для отображения содержимого в текстовом формате. Вы можете определить как версию в HTML, так и версию в plain-text для вашего сообщения:

```
/**  
 * Get the message content definition.  
 */  
public function content(): Content  
{  
    return new Content(  
        view: 'mail.orders.shipped',  
        text: 'mail.orders.shipped-text'  
    );  
}
```

Для большей ясности параметр `html` может быть использован в качестве псевдонима параметра `view`:

```
return new Content(  
    html: 'mail.orders.shipped',  
    text: 'mail.orders.shipped-text'  
);
```

## Данные шаблона

### Передача данных шаблону через публичные свойства

Как правило, вам нужно передать в шаблон некоторые данные, которые можно использовать при отображении HTML-кода электронного письма. Есть два способа сделать данные доступными для вашего шаблона. Во-первых, любое публичное свойство, определенное в вашем почтовом классе, будет автоматически доступно для шаблона. Так, например, можно передать данные в конструктор почтового класса и присвоить этим данные публичным свойствам, определенным в классе:

```
<?php
```

```

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Создать экземпляр нового сообщения.
     */
    public function __construct(
        public Order $order,
    ) {}

    /**
     * Получить содержимое сообщения
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
        );
    }
}

```

После того как данные были заданы как публичные свойства, они будут автоматически доступны в вашем шаблоне, поэтому вы можете получить к ним доступ так же, как и к любым другим данным в ваших шаблонах Blade:

```

<div>
    Price: {{ $order->price }}
</div>

```

## Передача данных шаблону через параметр `with`

Если вы хотите настроить формат данных вашего электронного письма перед их отправкой в шаблон, то вы можете вручную передать свои данные в шаблон с помощью параметра `with`. Как правило, вы по-прежнему будете передавать данные через конструктор почтового класса; однако, вы должны установить для этих

данных свойства `protected` или `private`, чтобы данные не были автоматически доступны для шаблона:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Создать экземпляр нового сообщения.
     */
    public function __construct(
        protected Order $order,
    ) {}

    /**
     * Получить содержимое сообщения
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
            with: [
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
            ],
        );
    }
}
```

После того как данные были переданы методу `with`, они автоматически станут доступны в вашем шаблоне, поэтому вы можете получить к ним доступ так же, как и к любым другим данным в ваших шаблонах Blade:

```
<div>
    Price: {{ $orderPrice }}
```

```
</div>
```

## Вложения

Для добавления вложений к электронному письму, вы добавляете их в массив, который возвращает метод `attachments` сообщения. Сначала вы можете добавить вложение, указав путь к файлу с использованием метода `fromPath` класса `Attachment`:

```
use Illuminate\Mail\Mailables\Attachment;

/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file'),
    ];
}
```

При прикреплении файлов к сообщению вы также можете указать отображаемое имя и/или MIME-тип, используя методы `as` и `withMime`:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf'),
    ];
}
```

## Прикрепление файлов с диска

Если вы сохранили файл на одном из [дисков файлового хранилища](#), то вы можете прикрепить его к электронному письму с помощью метода `fromStorage`:

```
/**  
 * Get the attachments for the message.  
 *  
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>  
 */  
public function attachments(): array  
{  
    return [  
        Attachment::fromStorage('/path/to/file'),  
    ];  
}
```

Конечно, вы также можете указать имя и MIME-тип вложения:

```
/**  
 * Get the attachments for the message.  
 *  
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>  
 */  
public function attachments(): array  
{  
    return [  
        Attachment::fromStorage('/path/to/file')  
            ->as('name.pdf')  
            ->withMime('application/pdf'),  
    ];  
}
```

Метод `fromStorageDisk` используется, если вам нужно указать диск хранения, отличный от вашего диска по умолчанию:

```
/**  
 * Get the attachments for the message.  
 *  
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>  
 */  
public function attachments(): array  
{  
    return [  
        Attachment::fromStorageDisk('s3', '/path/to/file')  
            ->as('name.pdf')  
    ];  
}
```

```
        ->withMime('application/pdf'),  
    ];  
}
```

## Вложения необработанных данных

Метод `fromData` используется для присоединения сырой строки байтов в качестве вложения. Например, вы можете использовать этот метод, если вы сгенерировали PDF-файл в памяти и хотите прикрепить его к электронному письму, не записывая его на диск. Метод `fromData` принимает замыкание, которое разрешает сырье байты данных, а также имя, которое следует присвоить вложению:

```
/**  
 * Get the attachments for the message.  
 *  
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>  
 */  
public function attachments(): array  
{  
    return [  
        Attachment::fromData(fn () => $this->pdf, 'Report.pdf')  
            ->withMime('application/pdf'),  
    ];  
}
```

## Встраиваемые вложения

Встраивание изображений в ваши электронные письма, как правило, обременительно; однако Laravel предлагает удобный способ прикреплять изображения к вашим письмам. Чтобы встроить изображение, используйте метод `embed` для переменной `$message` в вашем шаблоне электронной почты. Laravel автоматически делает переменную `$message` доступной для всех ваших шаблонов электронной почты, поэтому вам не нужно беспокоиться о ее передаче вручную:

```
<body>  
    Here is an image:  
  
      
</body>
```

Переменная `$message` недоступна в шаблонах текстовых сообщений, так как в текстовых сообщениях не используются встроенные вложения.

## Встраиваемые вложения необработанных данных

Если у вас уже есть строка необработанных данных изображения, которую вы хотите встроить в шаблон электронной почты, то вы можете вызвать метод `embedData` для переменной `$message`. При вызове метода `embedData` вам необходимо указать имя файла, которое должно быть присвоено встраиваемому изображению:

```
<body>
    Here is an image from raw data:

    
</body>
```

## Объекты, которые можно прикреплять

В большинстве случаев прикрепление файлов к сообщениям с указанием путей является достаточным, но во многих случаях объекты, которые можно прикреплять, уже представлены классами в вашем приложении. Например, если ваше приложение прикрепляет фотографию к сообщению, то в вашем приложении может существовать модель `Photo`, которая представляет эту фотографию. В таком случае было бы удобно просто передать модель `Photo` методу `attach`. Объекты, которые можно прикреплять, позволяют вам сделать именно это.

Для начала реализуйте интерфейс `Illuminate\Contracts\Mail\Attachable` для объекта, который будет являться прикрепляемым к сообщениям. Этот интерфейс предписывает, что ваш класс должен определить метод `toMailAttachment`, который возвращает экземпляр `Illuminate\Mail\Attachment`:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Mail\Attachable;
use Illuminate\Database\Eloquent\Model;
```

```
use Illuminate\Mail\Attachment;

class Photo extends Model implements Attachable
{
    /**
     * Get the attachable representation of the model.
     */
    public function toMailAttachment(): Attachment
    {
        return Attachment::fromPath('/path/to/file');
    }
}
```

После того как вы определите свой объект, который можно прикреплять, вы можете вернуть экземпляр этого объекта из метода `attachments`, когда создаете сообщение электронной почты:

```
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [$this->photo];
}
```

Конечно, данные вложения могут храниться на удаленном сервисе файлового хранения, таком как Amazon S3. Поэтому Laravel также позволяет создавать экземпляры вложений на основе данных, хранящихся на одном из дисков файловой системы вашего приложения:

```
// Создание вложения из файла на вашем основном диске...
return Attachment::fromStorage($this->path);

// Создание вложения из файла на конкретном диске...
return Attachment::fromStorageDisk('backblaze', $this->path);
```

Кроме того, вы можете создавать экземпляры вложений на основе данных, хранящихся в памяти. Для этого передайте замыкание методу `fromData`. Замыкание должно возвращать сырье данные, представляющие вложение:

```
return Attachment::fromData(fn () => $this->content, 'Имя фотографии');
```

Laravel также предоставляет дополнительные методы, которые вы можете использовать для настройки ваших вложений. Например, вы можете использовать методы `as` и `withMime` для настройки имени файла и MIME-типа:

```
return Attachment::fromPath('/путь/к/файлу')
    ->as('Имя фотографии')
    ->withMime('image/jpeg');
```

## Заголовки

Иногда вам может потребоваться прикрепить дополнительные заголовки к исходящему сообщению. Например, вам может потребоваться установить пользовательский `Message-Id` или другие произвольные текстовые заголовки.

Для этого определите метод `headers` в вашем классе для отправки электронной почты. Метод `headers` должен возвращать экземпляр класса `Illuminate\Mail\Mailables\Headers`. Этот класс принимает параметры `messageId`, `references` и `text`. Конечно, вы можете предоставить только те параметры, которые вам нужны для вашего конкретного сообщения:

```
use Illuminate\Mail\Mailables\Headers;

/**
 * Получить заголовки сообщения.
 */
public function headers(): Headers
{
    return new Headers(
        messageId: 'custom-message-id@example.com',
        references: ['previous-message@example.com'],
        text: [
            'X-Custom-Header' => 'Custom Value',
        ],
    );
}
```

```
use Illuminate\Mail\Mailables\Envelope;
```

```

/**
 * Получите конверт сообщения.
 *
 * @return \Illuminate\Mail\Mailables\Envelope
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Заказ отправлен',
        tags: ['shipment'],
        metadata: [
            'order_id' => $this->order->id,
        ],
    );
}

```

Если ваше приложение использует драйвер Mailgun, вы можете проконсультироваться с документацией Mailgun для получения дополнительной информации о [тегах](#) и [метаданных](#). Аналогично, вы можете проконсультироваться с документацией Postmark для получения дополнительной информации о поддержке [тегов](#) и [метаданных](#).

Если ваше приложение использует Amazon SES для отправки электронных писем, вы должны использовать метод `metadata` для добавления [тегов SES](#) к сообщению.

## Настройка Symfony Message

Функциональность почты Laravel основана на Symfony Mailer. Laravel позволяет вам зарегистрировать пользовательские обратные вызовы (замыкания), которые будут вызываться с экземпляром Symfony Message перед отправкой сообщения. Это дает вам возможность глубоко настраивать сообщение перед его отправкой. Для этого определите параметр `using` в вашем определении `Envelope`:

```

use Illuminate\Mail\Mailables\Envelope;
use Symfony\Component\Mime>Email;

/**
 * Получите конверт сообщения.
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Заказ отправлен',
        using: [

```

```
        function (Email $message) {
            // ...
        },
    ],
);
}
```

## # Отправления с разметкой Markdown

Почтовые сообщения с разметкой Markdown позволяют вам воспользоваться преимуществами предварительно созданных шаблонов и компонентов [почтовых уведомлений](#) в ваших почтовых рассылках. Поскольку сообщения написаны на Markdown, Laravel может отображать красивые, отзывчивые HTML-шаблоны для сообщений, а также автоматически генерировать их аналоги в виде простого текста.

## Генерация отправлений с разметкой Markdown

Чтобы сгенерировать почтовый класс с соответствующим шаблоном Markdown, вы можете использовать параметр `--markdown` в команде `make:mail` Artisan:

```
php artisan make:mail OrderShipped --markdown=mail.orders.shipped
```

Затем, при настройке определения `Content` внутри его метода `content`, используйте параметр `markdown` вместо параметра `view`:

```
/**
 * Get the message content definition.
 */
public function content(): Content
{
    return new Content(
        markdown: 'mail.orders.shipped',
        with: [
            'url' => $this->orderUrl,
        ],
    );
}
```

## Написание сообщений с разметкой Markdown

Почтовые сообщения Markdown используют комбинацию компонентов Blade и синтаксиса Markdown, которые позволяют легко создавать почтовые сообщения, используя предварительно созданные компоненты пользовательского интерфейса электронной почты Laravel:

```
<x-mail::message>
# Order Shipped
Your order has been shipped!
<x-mail::button :url="$url">
View Order
</x-mail::button>
Thanks,<br>
{{ config('app.name') }}
</x-mail::message>
```

Не используйте лишние отступы при написании писем Markdown. По стандартам Markdown парсеры будут отображать контент с отступом в виде блоков кода.

## Компонент Button

Компонент кнопки отображает ссылку на кнопку по центру. Компонент принимает два аргумента: `url` и необязательный `color`. Поддерживаемые цвета: `primary`, `success`, и `error`. Вы можете добавить к сообщению столько компонентов кнопки, сколько захотите:

```
<x-mail::button :url="$url" color="success">
View Order
</x-mail::button>
```

## Компонент Panel

Компонент панели отображает указанный блок текста на панели, цвет фона которой немного отличается от цвета остальной части сообщения. Это позволяет привлечь внимание к указанному блоку текста:

```
<x-mail::panel>
This is the panel content.
</x-mail::panel>
```

## Компонент Table

Компонент таблицы позволяет преобразовать таблицу Markdown в таблицу HTML. Компонент принимает в качестве содержимого таблицу Markdown. Выравнивание столбцов таблицы поддерживается с использованием синтаксиса выравнивания таблицы Markdown по умолчанию:

```
<x-mail::table>
| Laravel      | Table          | Example      |
| -----:       | :-----:       | -----:       |
| Col 2 is    | Centered      | $10          |
| Col 3 is    | Right-Aligned | $20          |
</x-mail::table>
```

## Изменение компонентов

Вы можете экспортить все почтовые компоненты Markdown в собственное приложение для настройки. Чтобы экспортить компоненты, используйте команду `vendor:publish` Artisan с параметром `--tag=laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

Эта команда опубликует почтовые компоненты Markdown в каталоге `resources/views/vendor/mail`. Каталог `mail` будет содержать каталог `html` и `text`, каждый из которых содержит соответствующие представления каждого доступного компонента. Вы можете настроить эти компоненты по своему усмотрению.

## Редактирование файла CSS

После экспорта компонентов в каталоге `resources/views/vendor/mail/html/themes` будет содержаться файл `default.css`. Вы можете отредактировать CSS в этом файле, и ваши стили будут автоматически преобразованы во встроенные стили CSS в HTML-представлениях ваших почтовых сообщений Markdown.

Если вы хотите создать совершенно новую тему для компонентов Laravel Markdown, вы можете поместить файл CSS в каталог `html/themes`. После присвоения имени и сохранения файла CSS обновите параметр `theme` в файле конфигурации вашего приложения `config/mail.php`, чтобы он соответствовал имени вашей новой темы.

Чтобы настроить тему для отдельного почтового сообщения, вы можете установить в свойстве `$theme` почтового класса имя темы, которое следует использовать при отправке этого почтового сообщения.

## # Отправка почты

Чтобы отправить сообщение, используйте метод [to фасада Mail](#). Метод `to` принимает адрес электронной почты, экземпляр пользователя или коллекцию пользователей. Если вы передаете объект или коллекцию объектов, почтовая программа будет автоматически использовать их свойства `email` и `name` при определении получателей электронной почты, поэтому убедитесь, что эти атрибуты доступны для ваших объектов. После того как вы указали своих получателей, вы можете передать экземпляр вашего почтового класса методу `send`:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * Отправить заказ.
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // Отправляем заказ ...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}
```

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

Вы не ограничены простым указанием получателей при отправке сообщения. Вы можете указать получателей `to`, `cc` и `bcc`, связав их соответствующие методы вместе:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

## Итерация списка получателей

Иногда требуется отправить почтовое сообщение списку получателей, перебирая массив получателей / адресов электронной почты. Однако, поскольку метод `to` добавляет адреса электронной почты к списку получателей почтового сообщения, каждая итерация цикла будет отправлять другое электронное письмо каждому предыдущему получателю. Следовательно, вы всегда должны повторно создавать почтовый экземпляр для каждого получателя:

```
foreach(['taylor@example.com', 'dries@example.com'] as $recipient) {
    Mail::to($recipient)->send(new OrderShipped($order));
}
```

## Указание драйвера при отправке почты

По умолчанию Laravel будет отправлять электронную почту, используя почтовую программу, настроенную как почтовую программу `default` в файле конфигурации вашего приложения `mail`. Однако вы можете использовать метод `mailer` для отправки сообщения с использованием определенной конфигурации почтовой программы:

```
Mail::mailer('postmark')
    ->to($request->user())
    ->send(new OrderShipped($order));
```

# Очередь почты

## Постановка сообщения в очередь почты

Поскольку отправка сообщений электронной почты может негативно повлиять на время отклика вашего приложения, многие разработчики ставят сообщения электронной почты в очередь для фоновой отправки. Laravel упрощает это с помощью встроенного [API унифицированной очереди](#). Чтобы поставить почтовое сообщение в очередь, используйте метод `queue` фасада `Mail` после указания получателей сообщения:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

Этот метод автоматически помещает задание в очередь, чтобы сообщение отправлялось в фоновом режиме. Перед использованием этого функционала вам необходимо [настроить очереди](#).

## Очередь отложенных сообщений

Если вы хотите отложить доставку электронного сообщения в очереди, вы можете использовать метод `later`. В качестве первого аргумента метод `later` принимает экземпляр `DateTime`, указывающий, когда сообщение должно быть отправлено:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

## Постановка сообщения в конкретную очередь почты

Поскольку все почтовые классы, сгенерированные с помощью команды `make:mail`, используют трейт `Illuminate\Bus\Queueable`, вы можете вызвать методы `onQueue` и `onConnection` для любого экземпляра почтового класса, что позволит вам указать соединение и имя очереди для сообщения:

```
$message = (new OrderShipped($order))
            ->onConnection('sq')
            ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

## Очередь почты, используемая по умолчанию

Если у вас есть почтовые классы, которые вы хотите всегда ставить в очередь, то вы можете реализовать контракт `ShouldQueue` для этого класса. Теперь, даже если вы вызовете метод `send` для отправки, почтовый класс все равно будет помещен в очередь, поскольку он содержит контракт:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    // ...
}
```

## Почтовые сообщения в очереди и транзакции в базе данных

Когда помещенные в очередь почтовые сообщения отправляются в рамках транзакций базы данных, они могут быть обработаны очередью до того, как транзакция базы данных будет зафиксирована. Когда это происходит, любые обновления, внесенные вами в модели или записи базы данных во время транзакции базы данных, могут еще не быть отражены в базе данных. Кроме того, любые модели или записи базы данных, созданные в рамках транзакции, могут не существовать в базе данных. Если ваше почтовое сообщение зависит от этих моделей, при обработке задания, отправляющего почтовое сообщение в очереди, могут возникнуть непредвиденные ошибки.

Если для параметра `after_commit` конфигурации вашего соединения с очередью задано значение `false`, то вы все равно можете указать, что конкретное почтовое сообщение в очереди должно быть отправлено после того, как все открытые

транзакции базы данных были зафиксированы, путем вызова метода `afterCommit` при отправке почтового сообщения:

```
Mail::to($request->user())->send(  
    (new OrderShipped($order))->afterCommit()  
)
```

В качестве альтернативы вы можете вызвать метод `afterCommit` из конструктора вашего почтового сообщения:

```
<?php  
  
namespace App\Mail;  
  
use Illuminate\Bus\Queueable;  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Mail\Mailable;  
use Illuminate\Queue\SerializesModels;  
  
class OrderShipped extends Mailable implements ShouldQueue  
{  
    use Queueable, SerializesModels;  
  
    /**  
     * Create a new message instance.  
     */  
    public function __construct()  
    {  
        $this->afterCommit();  
    }  
}
```

Чтобы больше узнать о том, как обойти эти проблемы, просмотрите документацию, касающуюся [заданий в очереди и транзакций базы данных](#).

## # Отображение отправлений

Иногда требуется получить HTML-содержимое почтового сообщения, не отправляя его. Для этого вы можете вызвать метод `render` почтового сообщения. Этот метод вернет проанализированное HTML-содержимое почтового сообщения в виде строки:

```
use App\Mail\InvoicePaid;  
use App\Models\Invoice;  
  
$invoice = Invoice::find(1);  
  
return (new InvoicePaid($invoice))->render();
```

## Предварительный просмотр отправлений в браузере

При разработке шаблона почтового сообщения удобно быстро просмотреть визуализированное почтовое сообщение в браузере как типичный шаблон Blade. По этой причине Laravel позволяет вам возвращать любое почтовое сообщение непосредственно из замыкания маршрута или контроллера. Когда почтовое сообщение возвратится, оно будет обработано и отображено в браузере, что позволит вам быстро просмотреть его дизайн без необходимости отправлять его на реальный адрес электронной почты:

```
Route::get('/mailable', function () {  
    $invoice = App\Models\Invoice::find(1);  
  
    return new App\Mail\InvoicePaid($invoice);  
});
```

## # Локализация отправлений

Laravel позволяет отправлять почтовые сообщения, используя язык, отличный от текущего языка запроса, и даже будет помнить его, если почта находится в очереди.

Для этого фасад `Mail` содержит метод `locale` для установки желаемого языка. Приложение изменит язык при анализе шаблона почтового сообщения, а затем вернется к предыдущему языку, когда анализ будет завершен:

```
Mail::to($request->user())->locale('es')->send(  
    new OrderShipped($order)  
)
```

## Предпочитаемые пользователем локализации

Иногда приложения хранят предпочтительный язык каждого пользователя.

Реализуя контракт `HasLocalePreference` в ваших моделях, вы можете указать Laravel использовать этот сохраненный язык при отправке почты:

```
use Illuminate\Contracts\Translation\HasLocalePreference;  
  
class User extends Model implements HasLocalePreference  
{  
    /**  
     * Получить предпочтаемую пользователем локализацию.  
     */  
    public function preferredLocale(): string  
    {  
        return $this->locale;  
    }  
}
```

После того как вы реализовали интерфейс, Laravel будет автоматически использовать предпочтительный язык при отправке уведомлений и почтовых сообщений модели. Следовательно, при использовании этого интерфейса нет необходимости вызывать метод `locale`:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

## # Тестирование

### Тестирование содержимого отправлений

Laravel предоставляет разнообразные методы для анализа структуры вашего класса для отправки электронной почты. Кроме того, Laravel предоставляет несколько удобных методов для проверки наличия ожидаемого содержимого в вашем классе для отправки электронной почты. Эти методы включают в себя: `assertSeeInHtml`, `assertDontSeeInHtml`, `assertSeeInOrderInHtml`, `assertSeeInText`,

`assertDontSeeInText`, `assertSeeInOrderInText`, `assertHasAttachment`, `assertHasAttachedData`, `assertHasAttachmentFromStorage` И `assertHasAttachmentFromStorageDisk`.

Как и следовало ожидать, утверждения «HTML» утверждают, что HTML-версия вашего почтового сообщения содержит переданную строку, в то время как утверждения «текст» утверждают, что текстовая версия вашего почтового сообщения содержит переданную строку:

Pest      PHPUnit

```
use App\Mail\InvoicePaid;
use App\Models\User;

test('mailable content', function () {
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertFrom('jeffrey@example.com');
    $mailable->assertTo('taylor@example.com');
    $mailable->assertHasCc('abigail@example.com');
    $mailable->assertHasBcc('victoria@example.com');
    $mailable->assertHasReplyTo('tyler@example.com');
    $mailable->assertHasSubject('Invoice Paid');
    $mailable->assertHasTag('example-tag');
    $mailable->assertHasMetadata('key', 'value');

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');
    $mailable->assertSeeInOrderInHtml(['Invoice Paid', 'Thanks']);

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInOrderInText(['Invoice Paid', 'Thanks']);

    $mailable->assertHasAttachment('/path/to/file');
    $mailable->assertHasAttachment(Attachment::fromPath('/path/to/file'));
    $mailable->assertHasAttachedData($pdfData, 'name.pdf', ['mime' => 'application/pdf']);
    $mailable->assertHasAttachmentFromStorage('/path/to/file', 'name.pdf', ['mime' => 'application/pdf']);
    $mailable->assertHasAttachmentFromStorageDisk('s3', '/path/to/file', 'name.pdf',
});
```

## Тестирование отправки почтовых сообщений

Мы рекомендуем тестировать содержимое ваших классов для отправки электронной почты отдельно от ваших тестов, которые утверждают, что определенное письмо было “отправлено” определенному пользователю. Обычно содержимое классов для отправки электронной почты не имеет отношения к коду, который вы тестируете, и достаточно просто утверждать, что Laravel был указан для отправки определенного класса для отправки электронной почты.

Вы можете использовать метод `fake` фасада `Mail`, чтобы предотвратить отправку электронных писем. После вызова метода `fake` фасада `Mail`, вы можете утверждать, что было указано отправить классы для отправки электронной почты пользователям и даже проверять данные, которые были получены классами для отправки электронной почты:

Pest      PHPUnit

```
<?php

use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;

test('orders can be shipped', function () {
    Mail::fake();

    // Выполните доставку заказа...

    // Утверждение, что ни одно письмо не было отправлено...
    Mail::assertNothingSent();

    // Утверждение, что было отправлено одно письмо...
    Mail::assertSent(OrderShipped::class);

    // Утверждение, что было отправлено два письма...
    Mail::assertSent(OrderShipped::class, 2);

    // Утверждение, что почтовое сообщение было отправлено на адрес электронной почты
    Mail::assertSent(OrderShipped::class, 'example@laravel.com');

    // Утверждение, что почтовое сообщение было отправлено на несколько адресов электронной почты
    Mail::assertSent(OrderShipped::class, ['example@laravel.com', ...]);

    // Утверждение, что почтовое сообщение не было отправлено...
    Mail::assertNotSent(AnotherMailable::class);

    // Утверждение, что всего было отправлено 3 почтовых сообщения...
})
```

```
    Mail::assertSentCount(3);  
});
```

Если вы отправляете отправку электронной почты в очередь в фоновом режиме, вам следует использовать метод `assertQueued` вместо `assertSent`:

```
Mail::assertQueued(OrderShipped::class);  
Mail::assertNotQueued(OrderShipped::class);  
Mail::assertNothingQueued();  
Mail::assertQueuedCount(3);
```

Вы можете передать замыкание в методы `assertSent`, `assertNotSent`, `assertQueued` или `assertNotQueued`, чтобы утверждать, что было отправлено письмо, которое соответствует определенному "тесту истинности". Если хотя бы одно письмо было отправлено и прошло указанный тест, то утверждение будет успешным:

```
Mail::assertSent(function (OrderShipped $mail) use ($order) {  
    return $mail->order->id === $order->id;  
});
```

При вызове методов проверки фасада `Mail`, принимаемый замыканием экземпляр класса для отправки электронной почты предоставляет полезные методы для анализа класса для отправки электронной почты:

```
Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($user) {  
    return $mail->hasTo($user->email) &&  
        $mail->hasCc('...') &&  
        $mail->hasBcc('...') &&  
        $mail->hasReplyTo('...') &&  
        $mail->hasFrom('...') &&  
        $mail->hasSubject('...');  
});
```

Экземпляр класса для отправки электронной почты также включает в себя несколько полезных методов для анализа вложений в классе для отправки электронной почты:

```
use Illuminate\Mail\Mailables\Attachment;

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromPath('/путь/к/файлу')
            ->as('name.pdf')
            ->withMime('application/pdf')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromStorageDisk('s3', '/путь/к/файлу')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($pdfData) {
    return $mail->hasAttachment(
        Attachment::fromData(fn () => $pdfData, 'name.pdf')
    );
});
```



Вы, возможно, заметили, что существует два метода для утверждения, что почта не была отправлена: `assertNotSent` и `assertNotQueued`. Иногда вам может потребоваться утверждать, что почта не была отправлена **или** поставлена в очередь. Для этого вы можете использовать методы `assertNothingOutgoing` и `assertNotOutgoing`:

```
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

## # Почта и локальная разработка

При разработке приложения для отправки электронной почты вы, вероятно, не захотите отправлять электронные письма на реальные адреса электронной почты. Laravel предлагает несколько способов «отключить» фактическую отправку электронных писем во время локальной разработки.

## Драйвер Log

Вместо того чтобы отправлять ваши электронные письма, почтовый драйвер [log](#) будет записывать все сообщения электронной почты в ваши файлы журналов для проверки. Обычно этот драйвер используется только во время локальной разработки. Для получения дополнительной информации о настройке вашего приложения для каждой среды ознакомьтесь с [документацией по конфигурации](#).

## HELO / Mailtrap / Mailpit

В качестве альтернативы вы можете использовать такую службу, как [HELO](#) или [Mailtrap](#) и драйвер [smtp](#), чтобы отправлять сообщения электронной почты в «фиктивный» почтовый ящик, где вы можете просмотреть их в настоящем почтовом клиенте. Этот подход имеет то преимущество, что позволяет вам фактически проверять окончательные электронные письма, непосредственно в почтовых службах.

Если вы используете [Laravel Sail](#), то вы можете предварительно просмотреть свои сообщения с помощью [Mailpit](#). Когда Sail запущен, вы можете получить доступ к интерфейсу Mailpit по адресу: <http://localhost:8025>.

## Использование глобального адреса to

Наконец, вы можете указать глобальный адрес «кому», вызвав метод [alwaysTo](#), предлагаемый фасадом Mail. Как правило, этот метод следует вызывать из метода [boot](#) одного из сервис-провайдеров вашего приложения:

```
use Illuminate\Support\Facades\Mail;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    if ($this->app->environment('local')) {
        Mail::alwaysTo('taylor@example.com');
    }
}
```

## # События

Laravel отправляет два события при отправке почтовых сообщений. Событие [MessageSending](#) отправляется до отправки сообщения, а событие [MessageSent](#) отправляется после отправки сообщения. Помните, что эти события отправляются при **отправке** почты, а не при ее постановке в очередь. Вы можете создать [слушателей](#) для этих событий в своем приложении:

```
use Illuminate\Mail\Events\MessageSending;
// use Illuminate\Mail\Events\MessageSent;

class LogMessage
{
    /**
     * Handle the given event.
     */
    public function handle(MessageSending $event): void
    {
        // ...
    }
}
```

## # Пользовательские транспорты

Laravel включает в себя разнообразные транспорты для отправки электронной почты; однако, возможно, вам захочется написать собственные для доставки электронной почты через другие службы, которые Laravel не поддерживает "из коробки". Для начала определите класс, который расширяет класс [Symfony\Component\Mailer\Transport\AbstractTransport](#). Затем реализуйте методы `doSend` и `__toString()` в вашем транспорте:

```
use MailchimpTransactional\ApiClient;
use Symfony\Component\Mailer\SentMessage;
use Symfony\Component\Mailer\Transport\AbstractTransport;
use Symfony\Component\Mime\Address;
use Symfony\Component\Mime\MessageConverter;

class MailchimpTransport extends AbstractTransport
{
    /**
     * Создайте новый экземпляр транспорта Mailchimp.
     */
    public function __construct(
        protected ApiClient $client,
    ) {
```

```

        parent::__construct();
    }

/**
 * {@inheritDoc}
 */
protected function doSend(SentMessage $message): void
{
    $email = MessageConverter::toEmail($message->getOriginalMessage());

    $this->client->messages->send(['message' => [
        'from_email' => $email->getFrom(),
        'to' => collect($email->getTo())->map(function (Address $email) {
            return ['email' => $email->getAddress(), 'type' => 'to'];
        })->all(),
        'subject' => $email->getSubject(),
        'text' => $email->getTextBody(),
    ]]);
}

/**
 * Получите строковое представление транспорта.
 */
public function __toString(): string
{
    return 'mailchimp';
}
}

```

После того как вы определите свой собственный транспорт, вы можете зарегистрировать его с помощью метода `extend`, предоставляемого фасадом `Mail`. Обычно это следует делать в методе `boot` служб-поставщиков вашего приложения, который находится в службе `AppServiceProvider`. Вам передается аргумент `$config`, который содержит массив конфигурации, определенной для отправителя почты в конфигурационном файле `config/mail.php` вашего приложения:

```

use App\Mail\MailchimpTransport;
use Illuminate\Support\Facades\Mail;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Mail::extend('mailchimp', function (array $config = []) {
        return new MailchimpTransport(/* ... */);
    });
}

```

```
});  
}  
}
```

После того как ваш собственный транспорт был определен и зарегистрирован, вы можете создать определение отправителя почты в конфигурационном файле вашего приложения `config/mail.php`, которое будет использовать новый транспорт:

```
'mailchimp' => [  
    'transport' => 'mailchimp',  
    // ...  
,
```

## Дополнительные транспорты Symfony

Laravel включает поддержку некоторых существующих транспортов для отправки почты, поддерживаемых Symfony, таких как Mailgun и Postmark. Однако, возможно, вам захочется расширить Laravel для поддержки дополнительных транспортов, поддерживаемых Symfony. Для этого вам нужно установить необходимый транспорт Symfony с помощью Composer и зарегистрировать его в Laravel. Например, вы можете установить и зарегистрировать Symfony mailer "Brevo" (ранее "Sendinblue"):

```
composer require symfony/brevo-mailer symfony/http-client
```

После установки пакета Brevo mailer вы можете добавить запись с вашими учетными данными API Brevo в конфигурационный файл `services` вашего приложения:

```
'brevo' => [  
    'key' => 'ваш-ключ-арі',  
,
```

Затем вы можете использовать метод `extend` фасада `Mail` для регистрации транспорта в Laravel. Обычно это следует делать в методе `boot` служб-поставщиков:

```
use Illuminate\Support\Facades\Mail;  
use Symfony\Component\Mailer\Bridge\Brevo\Transport\BrevoTransportFactory;  
use Symfony\Component\Mailer\Transport\Dsn;
```

```
/**  
 * Bootstrap any application services.  
 */  
public function boot(): void  
{  
    Mail::extend('brevo', function () {  
        return (new BrevoTransportFactory)->create(  
            new Dsn(  
                'brevo+api',  
                'default',  
                config('services.brevo.key')  
            )  
        );  
    });  
}
```

После регистрации вашего транспорта вы можете создать определение отправителя почты в конфигурационном файле вашего приложения [config/mail.php](#), которое будет использовать новый транспорт:

```
'brevo' => [  
    'transport' => 'brevo',  
    // ...  
,
```

# Уведомления

## # Введение

## # Генерация уведомлений

## # Отправка уведомлений

# Использование трейта Notifiable

# Использование фасада Notification

# Определение каналов доставки

# Очереди уведомлений

# Уведомления по запросу

## # Почтовые уведомления

# Формирование почтовых сообщений

# Изменение отправителя

# Изменение получателя

# Изменение темы сообщения

# Изменение почтового драйвера

# Изменение почтовых шаблонов

# Почтовые вложения

# Добавление тегов и метаданных

# Настройка сообщения Symfony

# Использование почтовых отправлений

# Предварительный просмотр почтовых уведомлений

## # Почтовые уведомления с разметкой Markdown

# Генерация сообщения

# Написание сообщения

# Изменение компонентов

## # Уведомления через канал database

# Предварительная подготовка базы данных

# Формирование уведомлений канала database

# Доступ к уведомлениям

# Отметка прочитанных уведомлений

## # Трансляция уведомлений

- # Предварительная подготовка трансляции
- # Формирование транслируемых уведомлений
- # Прослушивание транслируемых уведомлений

## # Уведомления через SMS

- # Предварительная подготовка канала SMS
- # Формирование уведомлений через SMS
- # Изменение номера отправителя
- # Добавление ссылки на клиента
- # Маршрутизация SMS-уведомлений

## # Уведомления через Slack

- # Предварительная подготовка канала Slack
- # Формирование уведомления через Slack
- # Взаимодействие в Slack
- # Маршрутизация уведомлений в Slack
- # Уведомление во внешние рабочие пространства Slack

## # Локализация уведомлений

- # Предпочитаемые пользователем локализации

## # Тестирование

### # События уведомления

### # Пользовательские каналы уведомлений

# # Введение

В дополнение к поддержке [отправки электронной почты](#), Laravel обеспечивает поддержку отправки уведомлений по различным каналам доставки, включая электронную почту, SMS (через [Vonage](#), бывший Nexmo) и [Slack](#). Кроме того, сообществом было создано множество [каналов уведомлений](#) для отправки уведомлений по десяткам различных каналов! Уведомления также могут храниться в базе данных, поэтому они могут быть отображены в вашем веб-интерфейсе.

Как правило, уведомления должны быть короткими информационными сообщениями, которые уведомляют пользователей о том, что произошло в вашем приложении. Например, если вы пишете приложение для выставления счетов, то

вы можете отправить своим пользователям уведомление «Счет оплачен» по каналам электронной почты и SMS.

## # Генерация уведомлений

В Laravel каждое уведомление представлено единым классом. Чтобы сгенерировать новое уведомление, используйте команду `make:notification Artisan`. Эта команда поместит новый класс уведомления в каталог `app/Notifications` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его:

```
php artisan make:notification InvoicePaid
```

Каждый класс уведомления содержит метод `via` и переменное количество методов формирования сообщений, таких как `toMail` или `toDatabase`, которые преобразуют уведомление в сообщение, адаптированное для этого конкретного канала.

## # Отправка уведомлений

### Использование трейта Notifiable

Уведомления могут быть отправлены двумя способами: с использованием метода `notify` трейта `Notifiable` или с помощью `фасада Notification`. Трейт `Notifiable` по умолчанию содержится в модели `App\Models\User` вашего приложения:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

Метод `notify`, предоставляемый этим трейтом, ожидает получить экземпляр уведомления:

```
use App\Notifications\InvoicePaid;  
  
$user->notify(new InvoicePaid($invoice));
```

Помните, что вы можете использовать трейт `Notifiable` в любой из ваших моделей. Вы не ограничены использованием его только в модели `User`.

## Использование фасада Notification

Как вариант, вы можете отправлять уведомления через [фасад Notification](#). Этот подход полезен при отправке уведомления нескольким уведомляемым объектам, например группе пользователей. Чтобы отправлять уведомления с помощью фасада, передайте все уведомляемые сущности и экземпляр уведомления методу `send`:

```
use Illuminate\Support\Facades\Notification;  
  
Notification::send($users, new InvoicePaid($invoice));
```

Вы также можете немедленно отправлять уведомления, используя метод `sendNow`. Этот метод немедленно отправит уведомление, даже если оно реализует интерфейс `ShouldQueue`:

```
Notification::sendNow($developers, new DeploymentCompleted($deployment));
```

## Определение каналов доставки

Каждый класс уведомлений имеет метод `via`, который определяет, по каким каналам будет доставлено уведомление. Уведомления можно отправлять по каналам `mail`, `database`, `broadcast`, `vonage` и `slack`.

Если вы хотите использовать другие каналы доставки, такие как Telegram или Pusher, то посетите веб-сайт сообщества [Laravel Notification Channels](#).

Метод `via` получает экземпляр `$notifiable`, представляющий экземпляр класса, которому отправляется уведомление. Вы можете использовать `$notifiable`, чтобы определить, по каким каналам должно доставляться уведомление:

```
/**
 * Получить каналы доставки уведомлений.
 *
 * @return array<int, string>
 */
public function via(object $notifiable): array
{
    return $notifiable->prefers_sms ? ['vonage'] : ['mail', 'database'];
}
```

## Очереди уведомлений

Перед отправкой уведомлений в очередь вы должны настроить и запустить [обработчик очереди](#).

Отправка уведомлений может занять время, особенно если каналу необходимо выполнить внешний вызов API для доставки уведомления. Чтобы ускорить время отклика вашего приложения, поместите ваше уведомление в очередь, добавив интерфейс `ShouldQueue` и трейт `Queueable` в ваш класс. Интерфейс и трейт уже импортированы для всех уведомлений, сгенерированных с помощью команды `make:notification`, поэтому вы можете сразу добавить их в свой класс уведомлений:

```
<?php

namespace App\Notifications;
```

```
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

После добавления интерфейса `ShouldQueue` к уведомлению вы можете отправить уведомление как обычно. Laravel обнаружит интерфейс `ShouldQueue` в классе и автоматически поставит в очередь доставку уведомления:

```
$user->notify(new InvoicePaid($invoice));
```

При постановке уведомлений в очередь для каждого получателя и комбинации каналов будет создана задача в очереди. Например, если ваше уведомление имеет три получателя и два канала, в очередь будет отправлено шесть задач.

## Отложенные уведомления

Если вы хотите отложить доставку уведомления, то вы можете вызвать метод `delay` экземпляра уведомления:

```
$delay = now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($delay));
```

Вы можете передать массив методу `delay`, чтобы указать величину задержки для определенных каналов:

```
$user->notify((new InvoicePaid($invoice))->delay([
    'mail' => now()->addMinutes(5),
    'sms' => now()->addMinutes(10),
]));
```

Как альтернативу, вы можете определить метод `withDelay` непосредственно в классе уведомления. Метод `withDelay` должен возвращать массив с именами каналов и значениями задержек:

```
/**  
 * Определение задержки доставки уведомления.  
 *  
 * @return array<string, \Illuminate\Support\Carbon>  
 */  
public function withDelay(object $notifiable): array  
{  
    return [  
        'mail' => now()->addMinutes(5),  
        'sms' => now()->addMinutes(10),  
        // Задержки для других каналов  
    ];  
}
```

Этот подход позволяет централизованно управлять задержками для разных каналов в одном месте, что упрощает поддержку и модификацию кода.

## Изменение соединения отложенных уведомлений

По умолчанию, уведомления, помещенные в очередь, будут использовать стандартное соединение очереди вашего приложения. Если вы хотите указать другое соединение, которое должно использоваться для конкретного уведомления, вы можете вызвать метод `onConnection` в конструкторе вашего уведомления:

```
<?php  
  
namespace App\Notifications;  
  
use Illuminate\Bus\Queueable;  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Notifications\Notification;  
  
class InvoicePaid extends Notification implements ShouldQueue  
{  
    use Queueable;  
  
    /**  
     * Создание нового экземпляра уведомления.  
     */  
    public function __construct()
```

```
{  
    $this->onConnection('redis');  
}  
}
```

Или, если вы хотите указать конкретное соединение с очередью, которое должно использоваться для каждого канала уведомлений, поддерживаемого уведомлением, вы можете определить метод `viaConnections` в вашем уведомлении. Этот метод должен возвращать массив пар имя канала / имя соединения с очередью:

```
/**  
 * Определение, какое соединение должно использоваться для каждого канала уведомлени  
 *  
 * @return array<string, string>  
 */  
public function viaConnections(): array  
{  
    return [  
        'mail' => 'redis',  
        'database' => 'sync',  
    ];  
}
```



## Изменение очереди канала уведомлений

Если вы хотите указать конкретную очередь, которая должна использоваться для каждого канала уведомления, поддерживаемого уведомлением, то вы можете определить метод `viaQueues` в своем уведомлении. Этот метод должен возвращать массив пар имя канала / имя очереди:

```
/**  
 * Определить, какие очереди следует использовать для каждого канала уведомления.  
 *  
 * @return array<string, string>  
 */  
public function viaQueues(): array  
{  
    return [  
        'mail' => 'mail-queue',  
        'slack' => 'slack-queue',  
    ];  
}
```

```
];
}
```

## Посредник для уведомлений в очереди

Уведомления в очереди могут определять промежуточное программное обеспечение [так же, как задания в очереди](#). Для начала определите метод `middleware` в своем классе уведомлений. Метод `middleware` получит переменные `$notifying` и `$channel`, которые позволят вам настроить возвращаемое промежуточное программное обеспечение в зависимости от места назначения уведомления:

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Get the middleware the notification job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(object $notifiable, string $channel)
{
    return match ($channel) {
        'email' => [new RateLimited('postmark')],
        'slack' => [new RateLimited('slack')],
        default => [],
    };
}
```

## Уведомления в очереди и транзакции в базе данных

Когда отложенные уведомления отправляются в транзакциях базы данных, они могут быть обработаны очередью до того, как транзакция базы данных будет зафиксирована. Когда это происходит, любые обновления, внесенные вами в модели или записи базы данных во время транзакции базы данных, могут еще не быть отражены в базе данных. Кроме того, любые модели или записи базы данных, созданные в рамках транзакции, могут не существовать в базе данных.

Если для параметра `after_commit` конфигурации вашего соединения с очередью установлено значение `false`, то вы все равно можете указать, что конкретное отложенное уведомление должно быть отправлено после того, как все открытые транзакции базы данных были зафиксированы, путем вызова метода `afterCommit` при отправке уведомления:

```
use App\Notifications\InvoicePaid;

$user->notify((new InvoicePaid($invoice))->afterCommit());
```

В качестве альтернативы вы можете вызвать метод `afterCommit` из конструктора вашего уведомления:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * Create a new notification instance.
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

Чтобы узнать больше о том, как обойти эти проблемы, просмотрите документацию, касающуюся [заданий в очереди и транзакций базы данных](#).

## Определение необходимости отправки уведомления в очередь

После того как уведомление из очереди было отправлено для фоновой обработки, оно обычно принимается работником очереди и отправляется предполагаемому получателю.

Однако, если вы хотите окончательно определить, следует ли отправлять уведомление в очереди после его обработки работником очереди, вы можете определить метод `shouldSend` в классе уведомлений. Если этот метод возвращает `false`, уведомление не будет отправлено:

```
/**
 * Определите, нужно ли отправлять уведомление.
 */
public function shouldSend(object $notifiable, string $channel): bool
{
    return $this->invoice->isPaid();
}
```

## Уведомления по запросу

По желанию можно отправить уведомление кому-то, кто не сохранен как «пользователь» вашего приложения. Используя метод `route` фасада `Notification`, вы можете указать информацию о маршрутизации специального уведомления перед отправкой уведомления:

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Support\Facades\Notification;

Notification::route('mail', 'taylor@example.com')
    ->route('vonage', '5555555555')
    ->route('slack', '#slack-channel')
    ->route('broadcast', [new Channel('channel-name')])
    ->notify(new InvoicePaid($invoice));
```

Если вы хотите указать имя получателя при отправке уведомления по запросу на маршрут `mail`, вы можете предоставить массив, содержащий адрес электронной почты в качестве ключа и имя в качестве значения первого элемента в массиве:

```
Notification::route('mail', [
    'barrett@example.com' => 'Barrett Blair',
])->notify(new InvoicePaid($invoice));
```

С помощью метода `routes` вы можете предоставить сразу несколько маршрутов для нескольких каналов уведомлений:

```
Notification::routes([
    'mail' => ['barrett@example.com' => 'Barrett Blair'],
    'vonage' => '5555555555',
])->notify(new InvoicePaid($invoice));
```

## # Почтовые уведомления

### Формирование почтовых сообщений

Если уведомление поддерживает отправку по электронной почте, то вы должны определить метод `toMail` в классе уведомления. Этот метод получит объект `$notifiable` и должен вернуть экземпляр `Illuminate\Notifications\Messages\MailMessage`.

Класс `MailMessage` содержит несколько простых методов, которые помогут вам создавать транзакционные сообщения электронной почты. Почтовые сообщения могут содержать строки текста, а также «призыв к действию». Давайте посмотрим на пример метода `toMail`:

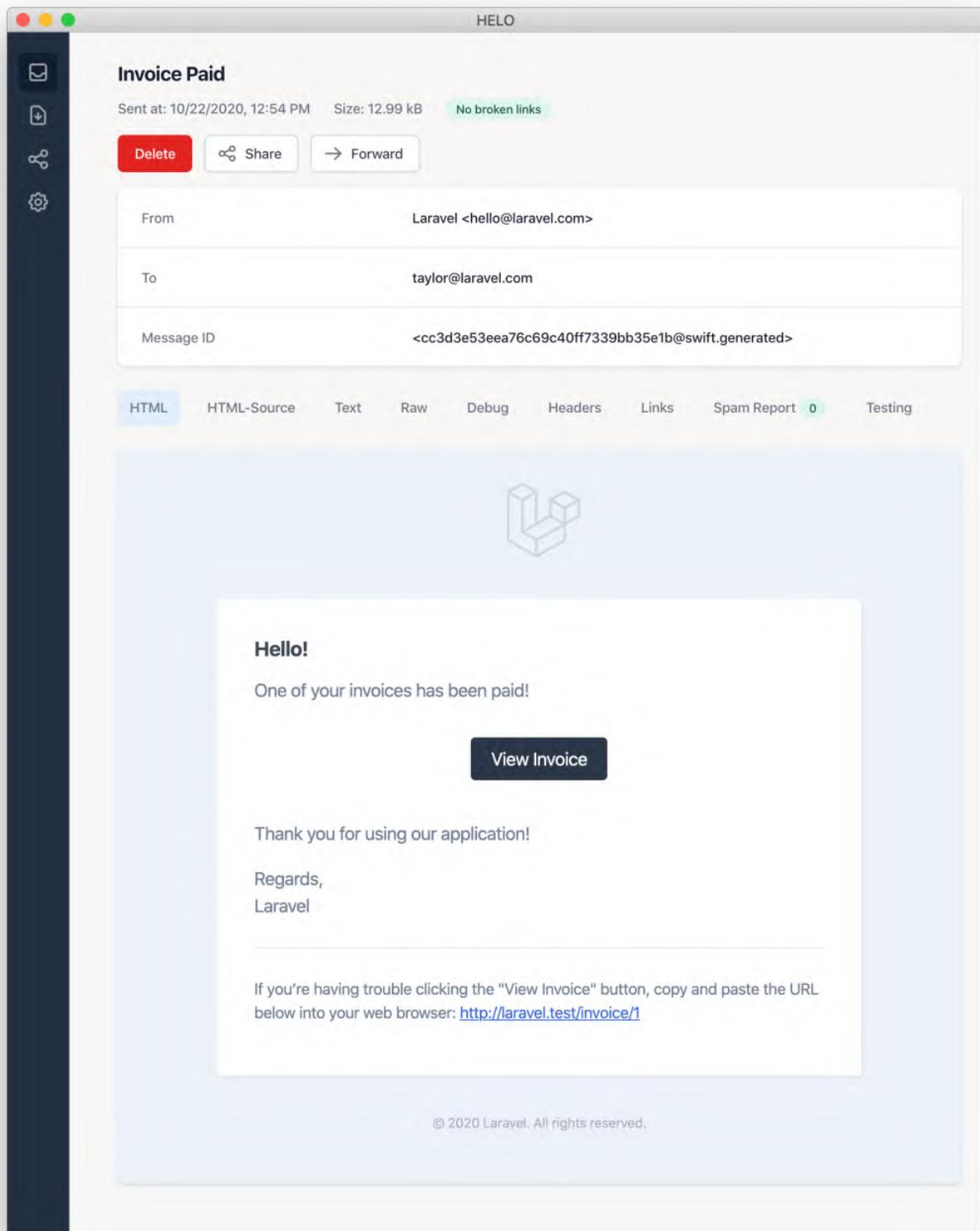
```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->greeting('Hello!')
        ->line('One of your invoices has been paid!')
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```

Обратите внимание, что мы используем `$this->invoice->id` в нашем методе `toMail`. Вы можете передать любые данные, которые необходимы

вашему уведомлению для генерации сообщения, в конструктор уведомления.

В этом примере мы регистрируем приветствие, строку текста, призыв к действию, а затем еще одну строку текста. Эти методы, предоставляемые объектом `MailMessage`, упрощают и ускоряют формирование небольших транзакционных электронных писем. Затем канал `mail` преобразует компоненты сообщения в красивый, отзывчивый HTML-шаблон сообщения электронной почты с аналогом в виде обычного текста. Вот пример электронного письма, созданного каналом `mail`:



Notification example

При отправке почтовых уведомлений не забудьте установить параметр `name` в вашем конфигурационном файле `config/app.php`. Это

значение будет использоваться в верхнем и нижнем колонтитулах ваших почтовых уведомлений.

## Сообщения об ошибках

Некоторые уведомления информируют пользователей об ошибках, таких как неудачный платеж по счету. Вы можете указать, что почтовое сообщение относится к ошибке, вызвав метод `error` при составлении вашего сообщения. При использовании метода `error` в почтовом сообщении кнопка действия будет красной, а не черной:

```
/**
 * Получить почтовое представление уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->error()
        ->subject('Invoice Payment Failed')
        ->line('...');
}
```

Этот подход помогает ясно передать пользователю, что сообщение содержит информацию об ошибке, повышая визуальную восприимчивость и понимание сообщения.

## Другие параметры формирования почтовых уведомлений

Вместо определения «строк» текста в классе уведомления, вы можете использовать метод `view`, чтобы указать собственный шаблон, который следует использовать для отображения почтового уведомления:

```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        'mail.invoice.paid', ['invoice' => $this->invoice]
```

```
    );
}
```

Вы можете определить текстовое содержимое для почтового сообщения, указав имя представления в качестве второго элемента массива, передаваемого методу `view`:

```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        ['mail.invoice.paid', 'mail.invoice.paid-text'],
        ['invoice' => $this->invoice]
    );
}
```

Или, если ваше сообщение содержит только простой текст, вы можете использовать метод `text`:

```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->text(
        'mail.invoice.paid-text', ['invoice' => $this->invoice]
    );
}
```

## Изменение отправителя

По умолчанию адрес отправителя электронного письма определяется в конфигурационном файле `config/mail.php`. Однако вы можете указать адрес отправителя для конкретного уведомления с помощью метода `from`:

```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
```

```
{  
    return (new MailMessage)  
        ->from('barrett@example.com', 'Barrett Blair')  
        ->line('...');  
}
```

## Изменение получателя

При отправке уведомлений по каналу `mail` система уведомлений автоматически ищет свойство `email` уведомляемого объекта. Вы можете указать, какой адрес электронной почты будет использоваться для доставки уведомления, определив метод `routeNotificationForMail` объекта уведомления:

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Foundation\Auth\User as Authenticatable;  
use Illuminate\Notifications\Notifiable;  
use Illuminate\Notifications\Notification;  
  
class User extends Authenticatable  
{  
    use Notifiable;  
  
    /**  
     * Маршрутизация уведомлений для почтового канала.  
     *  
     * @return array<string, string>|string  
     */  
    public function routeNotificationForMail(Notification $notification): array|string  
    {  
        // Вернуть только адрес электронной почты ...  
        return $this->email_address;  
  
        // Вернуть адрес электронной почты и имя ...  
        return [$this->email_address => $this->name];  
    }  
}
```

## Изменение темы сообщения

По умолчанию темой электронного письма является название класса уведомления в регистре «Title Case». Итак, если ваш класс уведомлений называется `InvoicePaid`, то темой электронного письма будет «Invoice Paid». Если вы хотите указать другую тему для сообщения, то вы можете вызвать метод `subject` при создании своего сообщения:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->subject('Notification Subject')  
        ->line('...');  
}
```

## Изменение почтового драйвера

По умолчанию уведомление по электронной почте будет отправлено с использованием почтового драйвера по умолчанию, определенной в конфигурационном файле `config/mail.php`. Однако вы можете указать другой почтовый драйвер во время выполнения, вызвав метод `mailer` при создании вашего сообщения:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->mailer('postmark')  
        ->line('...');  
}
```

## Изменение почтовых шаблонов

Вы можете изменить шаблон из HTML и обычного текста, используемый для почтовых уведомлений, опубликовав необходимые ресурсы уведомления. После выполнения этой команды шаблоны почтовых уведомлений будут расположены в каталоге `resources/views/vendor/notifications`:

```
php artisan vendor:publish --tag=laravel-notifications
```

## Почтовые вложения

Чтобы добавить вложения к почтовому уведомлению, используйте метод `attach` при создании сообщения. Метод `attach` принимает абсолютный путь к файлу в качестве своего первого аргумента:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->greeting('Hello!')  
        ->attach('/path/to/file');  
}
```

Метод `attach`, предоставляемый почтовыми сообщениями уведомлений, также принимает [объекты, прикрепляемые к сообщению](#). Пожалуйста, ознакомьтесь с подробной [документацией об объектах, прикрепляемых к сообщениям](#), чтобы узнать больше.

При прикреплении файлов к сообщению вы также можете указать отображаемое имя и / или MIME-тип, передав массив в качестве второго аргумента методу `attach`:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->greeting('Hello!')  
        ->attach('/path/to/file', [  
            'as' => 'name.pdf',  
            'mime' => 'application/pdf',  
        ]);  
}
```

```
]);  
}
```

В отличие от прикрепления файлов к почтовым отправлениям, вы не можете прикреплять файл непосредственно с диска файлового хранилища с помощью `attachFromStorage`. Лучше использовать метод `attach` с абсолютным путем к файлу на диске. В качестве альтернативы вы можете вернуть [отправление](#) из метода `toMail`:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;  
  
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): Mailable  
{  
    return (new InvoicePaidMailable($this->invoice))  
        ->to($notifiable->email)  
        ->attachFromStorage('/path/to/file');  
}
```

При необходимости к сообщению можно прикрепить несколько файлов, используя метод `attachMany`:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->greeting('Hello!')  
        ->attachMany([  
            '/path/to/forge.svg',  
            '/path/to/vapor.svg' => [  
                'as' => 'Logo.svg',  
                'mime' => 'image/svg+xml',  
            ],  
        ]);  
}
```

## Почтовые вложения необработанных данных

Метод `attachData` используется для присоединения необработанной строки в качестве вложения. При вызове метода `attachData` вы должны указать имя файла, которое должно быть присвоено вложению:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): Mailable  
{  
    return (new MailMessage)  
        ->greeting('Hello!')  
        ->attachData($this->pdf, 'name.pdf', [  
            'mime' => 'application/pdf',  
        ]);  
}
```

## Добавление тегов и метаданных

Некоторые сторонние почтовые провайдеры, такие как Mailgun и Postmark, поддерживают “теги” и “метаданные” сообщений, которые могут использоваться для группировки и отслеживания электронных писем, отправленных вашим приложением. Вы можете добавить теги и метаданные к электронному сообщению с помощью методов `tag` и `metadata`:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->greeting('Comment Upvoted!')  
        ->tag('upvote')  
        ->metadata('comment_id', $this->comment->id);  
}
```

Если ваше приложение использует драйвер Mailgun, вы можете обратиться к документации Mailgun для получения дополнительной информации о [тегах](#) и [метаданных](#). Аналогично, документацию Postmark можно также проконсультировать для получения информации о их поддержке [тегов](#) и [метаданных](#).

Если ваше приложение использует Amazon SES для отправки электронных писем, вы должны использовать метод `metadata` для прикрепления [тегов SES](#) к сообщению.

## Настройка сообщения Symfony

Метод `withSymfonyMessage` класса `MailMessage` позволяет зарегистрировать функцию обратного вызова, которая будет вызвана с экземпляром сообщения Symfony перед отправкой сообщения. Это дает вам возможность глубоко настроить сообщение перед его доставкой:

```
use Symfony\Component\Mime\Email;

/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
        ->withSymfonyMessage(function (Email $message) {
            $message->getHeaders()->addTextHeader(
                'Custom-Header', 'Header Value'
            );
        });
}
```

## Использование почтовых отправлений

При необходимости вы можете вернуть полный [объект почтового отправления](#) из метода `toMail` вашего уведомления. При возврате `Mailable` вместо `MailMessage` вам нужно будет указать получателя сообщения с помощью метода `to` объекта почтового отправления:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Mail\Mailable;

/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): Mailable
{
    return (new InvoicePaidMailable($this->invoice))
```

```
    ->to($notifiable->email);  
}
```

## Почтовые отправления и уведомления по запросу

Если вы отправляете [уведомление по запросу](#), то экземпляр `$notifiable`, переданный методу `toMail`, будет экземпляром `Illuminate\Notifications\AnonymousNotifiable`, содержащий метод `routeNotificationFor`, который можно использовать для получения адреса электронной почты для отправления уведомления по запросу:

```
use App\Mail\InvoicePaid as InvoicePaidMailable;  
use Illuminate\Notifications\AnonymousNotifiable;  
use Illuminate\Mail\Mailable;  
  
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): Mailable  
{  
    $address = $notifiable instanceof AnonymousNotifiable  
        ? $notifiable->routeNotificationFor('mail')  
        : $notifiable->email;  
  
    return (new InvoicePaidMailable($this->invoice))  
        ->to($address);  
}
```

## Предварительный просмотр почтовых уведомлений

При разработке шаблона почтового уведомления удобно быстро просмотреть визуализированное почтовое сообщение в браузере, как типичный шаблон Blade. По этой причине Laravel позволяет вам возвращать любое почтовое сообщение непосредственно из замыкания маршрута или контроллера. При возврате `MailMessage`, оно будет обработано и отображено в браузере, что позволит вам быстро просмотреть его дизайн без необходимости отправлять его на реальный адрес электронной почты:

```
use App\Models\Invoice;  
use App\Notifications\InvoicePaid;  
  
Route::get('/notification', function () {
```

```
$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))
    ->toMail($invoice->user);
});
```

## # Почтовые уведомления с разметкой Markdown

Почтовые уведомления с разметкой Markdown позволяют вам воспользоваться преимуществами предварительно созданых шаблонов почтовых уведомлений. Поскольку сообщения написаны на Markdown, Laravel может отображать красивые, отзывчивые HTML-шаблоны для сообщений, а также автоматически генерировать аналог в виде простого текста.

### Генерация сообщения

Чтобы сгенерировать уведомление с соответствующим шаблоном Markdown, вы можете использовать параметр `--markdown` команды `make:notification` Artisan:

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

Как и все другие почтовые уведомления, уведомления, использующие шаблоны Markdown, должны определять метод `toMail` в своем классе уведомлений. Однако вместо использования методов `line` и `action` для создания уведомления используйте метод `markdown`, чтобы указать имя шаблона Markdown, который следует использовать. Массив данных, который вы хотите сделать доступным для шаблона, может быть передан в качестве второго аргумента метода:

```
/**
 * Получить содержимое почтового уведомления.
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

# Написание сообщения

Почтовые уведомления Markdown используют комбинацию компонентов Blade и синтаксиса Markdown, которые позволяют легко создавать почтовые уведомления, используя предварительно созданные компоненты уведомлений Laravel:

```
<x-mail::message>
# Invoice Paid

Your invoice has been paid!

<x-mail::button :url="$url">
View Invoice
</x-mail::button>

Thanks,<br>
{{ config('app.name') }}
</x-mail::message>
```

## Компонент Button

Компонент кнопки отображает ссылку на кнопку по центру. Компонент принимает два аргумента: `url` и необязательный `color`. Поддерживаемые цвета: `primary`, `green`, и `red`. Вы можете добавить к уведомлению столько компонентов кнопки, сколько захотите:

```
<x-mail::button :url="$url" color="green">
View Invoice
</x-mail::button>
```

## Компонент Panel

Компонент панели отображает указанный блок текста на панели, цвет фона которой немного отличается от цвета остальной части сообщения. Это позволяет привлечь внимание к указанному блоку текста:

```
<x-mail::panel>
This is the panel content.
</x-mail::panel>
```

## Компонент Table

Компонент таблицы позволяет преобразовать таблицу Markdown в таблицу HTML. Компонент принимает в качестве содержимого таблицу Markdown. Выравнивание столбцов таблицы поддерживается с использованием синтаксиса выравнивания таблицы Markdown по умолчанию:

```
<x-mail::table>
| Laravel      | Table          | Example      |
| ----- | :-----: | -----: |
| Col 2 is    | Centered      | $10          |
| Col 3 is    | Right-Aligned | $20          |
</x-mail::table>
```

## Изменение компонентов

Вы можете экспортить все почтовые компоненты Markdown в собственное приложение для настройки. Чтобы экспортить компоненты, используйте команду `vendor:publish` Artisan с параметром `--tag=laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

Эта команда опубликует почтовые компоненты Markdown в каталоге `resources/views/vendor/mail`. Каталог `mail` будет содержать каталог `html` и `text`, каждый из которых содержит соответствующие представления каждого доступного компонента. Вы можете настроить эти компоненты по своему усмотрению.

## Редактирование файла CSS

После экспорта компонентов в каталоге `resources/views/vendor/mail/html/themes` будет содержаться файл `default.css`. Вы можете отредактировать CSS в этом файле, и ваши стили будут автоматически преобразованы во встроенные стили CSS в HTML-представлениях ваших почтовых сообщений Markdown.

Если вы хотите создать совершенно новую тему для компонентов Laravel Markdown, вы можете поместить файл CSS в каталог `html/themes`. После присвоения имени и сохранения файла CSS обновите параметр `theme` в файле конфигурации вашего приложения `config/mail.php`, чтобы он соответствовал имени вашей новой темы.

Чтобы настроить тему для отдельного уведомления, вы можете вызвать метод `theme` при создании почтового сообщения уведомления. Метод `theme` принимает имя темы, которая должна использоваться при отправке уведомления:

```
/**  
 * Получить содержимое почтового уведомления.  
 */  
public function toMail(object $notifiable): MailMessage  
{  
    return (new MailMessage)  
        ->theme('invoice')  
        ->subject('Invoice Paid')  
        ->markdown('mail.invoice.paid', ['url' => $url]);  
}
```

## # Уведомления через канал database

### Предварительная подготовка базы данных

Канал уведомлений `database` хранит информацию уведомления в таблице базы данных. Эта таблица будет содержать такую информацию, как тип уведомления, а также JSON-структуру данных, которая описывает уведомление.

Вы можете запросить таблицу, чтобы отобразить уведомления в пользовательском интерфейсе вашего приложения. Но прежде чем вы сможете это сделать, вам нужно будет создать таблицу базы данных для хранения ваших уведомлений. Вы можете использовать команду `make:notifications-table` для создания [миграции](#) с необходимой схемой таблицы:

```
php artisan make:notifications-table
```

```
php artisan migrate
```

Если ваши модели с уведомлениями используют [UUID или ULID в качестве первичных ключей](#), вы должны заменить метод `morphs` на `uuidMorphs` или `ulidMorphs` в миграции таблицы уведомлений.

# Формирование уведомлений канала database

Чтобы уведомление было сохранено в таблице базы данных, вы должны определить метод `toDatabase` или `toArray` в классе уведомления. Каждый из этих методов получает объект `$notifiable` и должен возвращать простой массив PHP. Возвращенный массив будет закодирован как JSON и сохранен в столбце `data` вашей таблицы `notifications`. Давайте посмотрим на пример метода `toArray`:

```
/**  
 * Получить массив данных уведомления.  
 *  
 * @return array<string, mixed>  
 */  
public function toArray(object $notifiable): array  
{  
    return [  
        'invoice_id' => $this->invoice->id,  
        'amount' => $this->invoice->amount,  
    ];  
}
```

Когда уведомление сохраняется в базе данных вашего приложения, столбец `type` заполняется именем класса уведомления. Однако вы можете настроить это поведение, определив метод `databaseType` в вашем классе уведомления:

```
/**  
 * Получить тип уведомления для базы данных.  
 *  
 * @return string  
 */  
public function databaseType(object $notifiable): string  
{  
    return 'invoice-paid';  
}
```

Этот метод позволяет вам устанавливать пользовательский тип уведомления, который будет сохранен в столбце `type` в таблице уведомлений базы данных. Это может быть полезно для более удобного отслеживания и фильтрации уведомлений по типу.

## Методы `toDatabase` и `toArray`

Метод `toArray` также используется каналом `broadcast`, чтобы определить, какие данные транслировать в JavaScript-приложение на клиентской стороне. Если вы хотите иметь два разных массива данных для каналов `database` и `broadcast`, то вы должны определить метод `toDatabase` вместо метода `toArray`.

## Доступ к уведомлениям

После сохранения уведомления в базу данных, вам понадобится удобный способ доступа к нему из уведомляемых объектов. Трейт `Illuminate\Notifications\Notifiable`, который по умолчанию расположен в модели `App\Models\User` Laravel, содержит отношение `notifications` Eloquent, возвращающее уведомления для объекта. Вы можете обратиться к этому методу, как и к любому другому отношению Eloquent, чтобы получить уведомления. По умолчанию уведомления будут упорядочены по столбцу `created_at` временной метки, причем самые последние уведомления будут помещены в начало коллекции:

```
$user = App\Models\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

Для получения только «непрочитанных» уведомлений, используйте отношение `unreadNotifications`. Опять же, эти уведомления будут упорядочены по столбцу `created_at` временной метки с самыми последними уведомлениями в начале коллекции:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

Чтобы получить доступ к уведомлениям в JavaScript-приложении на клиентской стороне, вы должны определить контроллер уведомлений для своего приложения, который возвращает уведомления для уведомляемого объекта, такого

как текущий пользователь. Затем вы можете сделать HTTP-запрос к URL-адресу этого контроллера из своего JavaScript-приложения на клиентской стороне.

## Отметка прочитанных уведомлений

По желанию можно пометить уведомление как «прочитанное», когда пользователь его просматривает. Трейт `Illuminate\Notifications\Notifiable` содержит метод `markAsRead`, который обновляет столбец `read_at` записи уведомления в базе данных:

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

Однако вместо того, чтобы перебирать каждое уведомление, вы можете использовать метод `markAsRead` непосредственно для коллекции уведомлений:

```
$user->unreadNotifications->markAsRead();
```

Вы также можете выполнить запрос массового обновления, чтобы пометить все уведомления как прочитанные, не извлекая их из базы данных:

```
$user = App\Models\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

Вы можете полностью удалить уведомления из таблицы, используя метод `delete`:

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

## # Трансляция уведомлений

# Предварительная подготовка трансляции

Перед трансляцией уведомлений вы должны настроить и ознакомиться со службами [трансляции событий](#) Laravel. Трансляция событий – способ реагирования на серверные события Laravel из своего JavaScript-приложения на клиентской стороне.

## Формирование транслируемых уведомлений

Канал `broadcast` транслирует уведомления с использованием служб [трансляции событий](#) Laravel, что позволяет вашему JavaScript-приложению на клиентской стороне улавливать уведомления в режиме реального времени. Если уведомление поддерживает трансляцию, то вы должны определить метод `toBroadcast` в классе уведомления. Этот метод получит объект `$notifiable` и должен вернуть экземпляр `BroadcastMessage`. Если метод `toBroadcast` не существует, то метод `toArray` будет использоваться для сбора данных, которые следует транслировать. Возвращенные данные будут закодированы как JSON и переданы вашему JavaScript-приложению на клиентской стороне. Давайте посмотрим на пример метода `toBroadcast`:

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Получить содержимое транслируемого уведомления.
 */
public function toBroadcast(object $notifiable): BroadcastMessage
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

## Конфигурирование очереди трансляции

Все транслируемые уведомления ставятся в очередь для трансляции. Если вы хотите изменить соединение очереди или имя очереди, которое используется для постановки в очередь трансляции, то вы можете использовать методы `onConnection` и `onQueue` экземпляра `BroadcastMessage`:

```
return (new BroadcastMessage($data))
    ->onConnection('sq')
```

```
->onQueue('broadcasts');
```

## Изменение типа транслируемого уведомления

В дополнение к указанным вами данным все транслируемые уведомления также имеют поле `type`, содержащее полное имя класса уведомления. Если вы хотите изменить `type` уведомления, то вы можете определить метод `broadcastType` в классе уведомления:

```
/**  
 * Получить тип транслируемого уведомления.  
 */  
public function broadcastType(): string  
{  
    return 'broadcast.message';  
}
```

## Прослушивание транслируемых уведомлений

Уведомления будут транслироваться по частному каналу, в формате с использованием соглашения `{notifiable}.{id}`. Итак, если вы отправляете уведомление экземпляру `App\Models\User` с идентификатором `1`, то уведомление будет транслироваться по частному каналу `App.Models.User.1`. При использовании [Laravel Echo](#) вы можете легко прослушивать уведомления канала, используя метод `notification`:

```
Echo.private('App.Models.User.' + userId)  
  .notification((notification) => {  
    console.log(notification.type);  
  });
```

## Изменение канала транслируемого уведомления

Если вы хотите изменить канал, на котором транслируются уведомления объекта, то вы можете определить метод `receivesBroadcastNotificationsOn` объекта уведомления:

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

```
use Illuminate\\Broadcasting\\PrivateChannel;
use Illuminate\\Foundation\\Auth\\User as Authenticatable;
use Illuminate\\Notifications\\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Каналы, по которым пользователь получает рассылку уведомлений.
     */
    public function receivesBroadcastNotificationsOn(): string
    {
        return 'users.'.$this->id;
    }
}
```

## # Уведомления через SMS

### Предварительная подготовка канала SMS

Отправка SMS-уведомлений в Laravel обеспечивается [Vonage](#) (бывший Nexmo). Прежде чем вы сможете отправлять уведомления через Vonage, вам необходимо установить пакеты `laravel/vonage-notification-channel` и `guzzlehttp/guzzle`:

```
composer require laravel/vonage-notification-channel guzzlehttp/guzzle
```

Пакет включает в себя [файл конфигурации](#). Однако вам не обязательно экспортировать этот файл конфигурации в ваше собственное приложение. Вы можете просто использовать переменные окружения `VONAGE_KEY` и `VONAGE_SECRET` для определения ваших публичного и секретного ключей Vonage.

После определения ваших ключей, вы должны установить переменную окружения `VONAGE_SMS_FROM`, которая определяет номер телефона, с которого по умолчанию будут отправляться ваши SMS-сообщения. Вы можете сгенерировать этот номер телефона в панели управления Vonage:

`VONAGE_SMS_FROM=15556666666`

# Формирование уведомлений через SMS

Если уведомление поддерживает отправку в виде SMS, то вы должны определить метод `toVonage` в классе уведомлений. Этот метод получит объект `$notifiable` и должен вернуть экземпляр `Illuminate\Notifications\Messages\NexmoMessage`:

```
/**  
 * Получить SMS-представление уведомления.  
 */  
public function toVonage(object $notifiable): VonageMessage  
{  
    return (new VonageMessage)  
        ->content('Your SMS message content');  
}
```

## Содержимое Unicode

Если ваше SMS-сообщение будет содержать символы Unicode, то вы должны вызвать метод `unicode` при создании экземпляра `VonageMessage`:

```
use Illuminate\Notifications\Messages\VonageMessage;  
  
/**  
 * Получить SMS-представление уведомления.  
 */  
public function toVonage(object $notifiable): VonageMessage  
{  
    return (new VonageMessage)  
        ->content('Your unicode message')  
        ->unicode();  
}
```

## Изменение номера отправителя

Если вы хотите отправить уведомление с номера телефона, который отличается от номера телефона, указанного в вашем файле `config/services.php`, то вы можете вызвать метод `from` экземпляра `VonageMessage`:

```
/**  
 * Получить Vonage / SMS-представление уведомления.  
 *  
 * @param mixed $notifiable
```

```

 * @return NexmoMessage
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->content('Your SMS message content')
        ->from('15554443333');
}

```

## Добавление ссылки на клиента

Если вы хотите отслеживать затраты на пользователя, команду или клиента, вы можете добавить в уведомление “ссылку на клиента”. Vonage позволит вам создавать отчеты, используя эту ссылку клиента, чтобы вы могли лучше понять использование SMS конкретным клиентом. Ссылка на клиента может быть любой строкой до 40 символов:

```

 /**
 * Получить Vonage / SMS-представление уведомления.
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
        ->clientReference((string) $notifiable->id)
        ->content('Your SMS message content');
}

```

## Маршрутизация SMS-уведомлений

Для отправки уведомления с использованием Vonage на необходимый номер телефона, определите метод `routeNotificationForVonage` вашего уведомляемого объекта:

```

<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{

```

```
use Notifiable;

/**
 * Маршрутизация уведомлений для канала Vonage.
 */
public function routeNotificationForVonage(Notification $notification): string
{
    return $this->phone_number;
}

}
```

## # Уведомления через Slack

### Предварительная подготовка канала Slack

Прежде чем вы сможете отправлять уведомления через Slack, вы должны установить канал уведомлений Slack через Composer:

```
composer require laravel/slack-notification-channel
```

Дополнительно, вы должны создать [Slack приложение](#) для вашего рабочего пространства Slack.

Если вам нужно отправлять уведомления только в то же рабочее пространство Slack, в котором создано приложение, убедитесь, что ваше приложение имеет разрешения `chat:write`, `chat:write.public` и `chat:write.customize`. Если вы хотите отправлять сообщения в качестве приложения Slack, вам следует убедиться, что ваше приложение также имеет область действия `chat:write:bot`. Эти разрешения можно добавить на вкладке “OAuth & Permissions” в управлении приложениями Slack.

Далее, скопируйте “Bot User OAuth Token” вашего приложения и поместите его в массив конфигурации `slack` в файле конфигурации `services.php` вашего приложения. Этот токен можно найти на вкладке “OAuth & Permissions” в Slack:

```
'slack' => [
    'notifications' => [
        'bot_user_oauth_token' => env('SLACK_BOT_USER_OAUTH_TOKEN'),
        'channel' => env('SLACK_BOT_USER_DEFAULT_CHANNEL'),
```

```
],  
],
```

## Распространение приложения Slack

Если ваше приложение будет отправлять уведомления во внешние рабочие пространства Slack, которые принадлежат пользователям вашего приложения, вам нужно будет “распространить” ваше приложение через Slack. Управление распространением приложения можно осуществить на вкладке “Manage Distribution” вашего приложения в Slack. После того, как ваше приложение будет распространено, вы можете использовать [Socialite](#), чтобы [получать токены Slack Bot](#) от имени пользователей вашего приложения.

## Формирование уведомления через Slack

Если уведомление поддерживает отправку в виде сообщения Slack, вы должны определить метод `toSlack` в классе уведомления. Этот метод получает сущность `$notifiable` и должен возвращать экземпляр `Illuminate\Notifications\Slack\SlackMessage`. Вы можете создавать богатые уведомления, используя [API Block Kit Slack](#). Следующий пример может быть пред просмотрен в [Block Kit Builder Slack](#):

```
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;  
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;  
use Illuminate\Notifications\Slack\BlockKit\Composites\ConfirmObject;  
use Illuminate\Notifications\Slack\SlackMessage;  
  
/**  
 * Получить представление Slack-уведомления.  
 */  
public function toSlack(object $notifiable): SlackMessage  
{  
    return (new SlackMessage)  
        ->text('One of your invoices has been paid!')  
        ->headerBlock('Invoice Paid')  
        ->contextBlock(function (ContextBlock $block) {  
            $block->text('Customer #1234');  
        })  
        ->sectionBlock(function (SectionBlock $block) {  
            $block->text('An invoice has been paid.');//  
            $block->field('*Invoice No:*\\n1000')->markdown();  
            $block->field('*Invoice Recipient:*\\ntaylor@laravel.com')->markdown()  
        })
```

```

->dividerBlock()
->sectionBlock(function (SectionBlock $block) {
    $block->text('Congratulations!');
});
}

```

## Использование шаблона Block Kit Builder Slack

Вместо использования методов построителя сообщений для создания сообщения Block Kit вы можете передать необработанный JSON, сгенерированный Block Kit Builder Slack, методу `usingBlockKitTemplate`:

```

use Illuminate\Notifications\Slack\SlackMessage;
use Illuminate\Support\Str;

/**
 * Get the Slack representation of the notification.
 */
public function toSlack(object $notifiable): SlackMessage
{
    $template = <<<JSON
    {
        "blocks": [
            {
                "type": "header",
                "text": {
                    "type": "plain_text",
                    "text": "Team Announcement"
                }
            },
            {
                "type": "section",
                "text": {
                    "type": "plain_text",
                    "text": "We are hiring!"
                }
            }
        ]
    }
    JSON;

    return (new SlackMessage)
        ->usingBlockKitTemplate($template);
}

```

# Взаимодействие в Slack

Система уведомлений Block Kit Slack предлагает мощные функции для [обработки взаимодействия пользователя](#). Чтобы использовать эти функции, ваше приложение Slack должно иметь включенную функцию "Interactivity" и настроенный "Request URL", который указывает на URL, обслуживаемый вашим приложением. Эти настройки можно управлять на вкладке "Interactivity & Shortcuts" в управлении приложениями Slack.

В следующем примере, который использует метод `actionsBlock`, Slack отправит `POST` запрос на ваш "Request URL" с полезной нагрузкой, содержащей информацию о пользователе Slack, который нажал на кнопку, идентификатор нажатой кнопки и дополнительную информацию. Ваше приложение может затем определить, какое действие следует предпринять на основе полученной полезной нагрузки. Также вы должны [проверить подлинность запроса](#), чтобы убедиться, что он был сделан Slack:

```
use Illuminate\Notifications\Slack\BlockKit\Blocks\ActionsBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;
use Illuminate\Notifications\Slack\SlackMessage;

/**
 * Получить представление Slack-уведомления.
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
        ->text('One of your invoices has been paid!')
        ->headerBlock('Invoice Paid')
        ->contextBlock(function (ContextBlock $block) {
            $block->text('Customer #1234');
        })
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('An invoice has been paid.');
        })
        ->actionsBlock(function (ActionsBlock $block) {
            // ID defaults to "button_acknowledge_invoice"...
            $block->button('Acknowledge Invoice')->primary();

            // Manually configure the ID...
            $block->button('Deny')->danger()->id('deny_invoice');
        });
}
```

## Модальные окна подтверждения

Если вы хотите, чтобы пользователи подтверждали действие перед его выполнением, вы можете использовать метод `confirm` при определении вашей кнопки. Метод `confirm` принимает сообщение и замыкание, которое получает экземпляр `ConfirmObject`:

```
use Illuminate\Notifications\Slack\BlockKit\Blocks\ActionsBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\ContextBlock;
use Illuminate\Notifications\Slack\BlockKit\Blocks\SectionBlock;
use Illuminate\Notifications\Slack\BlockKit\Composites\ConfirmObject;
use Illuminate\Notifications\Slack\SlackMessage;

/**
 * Получить представление Slack-уведомления.
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
        ->text('One of your invoices has been paid!')
        ->headerBlock('Invoice Paid')
        ->contextBlock(function (ContextBlock $block) {
            $block->text('Customer #1234');
        })
        ->sectionBlock(function (SectionBlock $block) {
            $block->text('An invoice has been paid.');
        })
        ->actionsBlock(function (ActionsBlock $block) {
            $block->button('Acknowledge Invoice')
                ->primary()
                ->confirm(
                    'Acknowledge the payment and send a thank you email?',
                    function (ConfirmObject $dialog) {
                        $dialog->confirm('Yes');
                        $dialog->deny('No');
                    }
                );
        });
}
```

## Просмотр структуры блоков Slack

Если вы хотите быстро проверить структуру блоков, которые вы создали, вы можете использовать метод `dd` в экземпляре `SlackMessage`. Метод `dd` сгенерирует и выведет URL-адрес для [Block Kit Builder Slack](#), который отображает

предварительный просмотр полезной нагрузки и уведомления в вашем браузере. Вы можете передать `true` методу `dd`, чтобы вывести сырую полезную нагрузку:

```
return (new SlackMessage)
    ->text('Один из ваших счетов оплачен!')
    ->headerBlock('Счет Оплачен')
    ->dd(); // Это вызовет просмотр блоков в Block Kit Builder
```

Этот метод особенно полезен во время разработки, так как позволяет быстро и удобно проверить внешний вид и структуру ваших уведомлений в Slack, не отправляя их на самом деле.

## Маршрутизация уведомлений в Slack

Чтобы направлять уведомления Slack в соответствующую команду Slack и канал, определите метод `routeNotificationForSlack` в вашей модели, уведомляемой событиями. Этот метод может возвращать одно из трех значений:

- `null` – что означает использование маршрутизации к каналу, настроенному в самом уведомлении. Вы можете использовать метод `to` при создании вашего `SlackMessage` для настройки канала в уведомлении.
- Строку, указывающую Slack канал, куда следует отправить уведомление, например, `#support-channel`.
- Экземпляр `SlackRoute`, который позволяет вам указать OAuth токен и имя канала, например, `SlackRoute::make($this->slack_channel, $this->slack_token)`. Этот метод следует использовать для отправки уведомлений во внешние рабочие пространства.

Например, возвращение `#support-channel` из метода `routeNotificationForSlack` отправит уведомление в канал `#support-channel` рабочего пространства, связанного с OAuth токеном Bot User, расположенным в файле конфигурации `services.php` вашего приложения:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
```

```
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Маршрутизация уведомлений для канала Slack.
     */
    public function routeNotificationForSlack(Notification $notification): string
    {
        return '#support-channel';
    }
}
```

## Уведомление во внешние рабочие пространства Slack

Прежде чем отправлять уведомления во внешние рабочие пространства Slack, ваше приложение Slack должно быть [распространено](#).

Конечно, часто вам захочется отправлять уведомления в рабочие пространства Slack, которые принадлежат пользователям вашего приложения. Для этого сначала вам потребуется получить OAuth-токен Slack для пользователя. К счастью, [Laravel Socialite](#) включает драйвер Slack, который позволит вам легко аутентифицировать пользователей вашего приложения в Slack и [получить токен бота](#).

После получения токена бота и его сохранения в базе данных вашего приложения, вы можете использовать метод `SlackRoute::make` для направления уведомления в рабочее пространство пользователя. Кроме того, ваше приложение, скорее всего, должно предоставить пользователю возможность указать, в какой канал следует отправлять уведомления:

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
```

```
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Slack\SlackRoute;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Маршрутизация уведомлений для канала Slack.
     *
     * @param \Illuminate\Notifications\Notification $notification
     * @return string
     */
    public function routeNotificationForSlack(Notification $notification): mixed
    {
        return SlackRoute::make($this->slack_channel, $this->slack_token);
    }
}
```

## # Локализация уведомлений

Laravel позволяет отправлять уведомления, используя язык, отличный от текущего языка запроса, и даже будет помнить его, если уведомление находится в очереди.

Для этого класс `\Illuminate\Notifications\Notification` содержит метод `locale` для установки желаемого языка. Приложение изменит язык при анализе уведомления, а затем вернется к предыдущему языку, когда анализ будет завершен:

```
$user->notify((new InvoicePaid($invoice))->locale('es'));
```

Локализация нескольких уведомляемых записей также доступна через фасад `Notification`:

```
Notification::locale('es')->send(
    $users, new InvoicePaid($invoice)
);
```

## Предпочитаемые пользователем локализации

Иногда приложения хранят предпочтительный язык каждого пользователя. Реализуя контракт `HasLocalePreference` в вашей уведомляемой модели, вы можете указать Laravel использовать этот сохраненный язык при отправке уведомления:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Получить предпочтаемую пользователем локализацию.
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

После того как вы реализовали интерфейс, Laravel будет автоматически использовать предпочтительный язык при отправке уведомлений и почтовых сообщений модели. Следовательно, при использовании этого интерфейса нет необходимости вызывать метод `locale`:

```
$user->notify(new InvoicePaid($invoice));
```

## # Тестирование

Вы можете использовать метод `fake` фасада `Notification`, чтобы предотвратить отправку уведомлений. Как правило, отправка уведомлений не имеет отношения к коду, который вы фактически тестируете. Вероятно, будет достаточно просто утверждать, что Laravel получил инструкцию отправить определенное уведомление.

После вызова метода `fake` фасада `Notification`, вы можете проверить, было ли передано инструкции отправить уведомления пользователям, и даже проверить данные, полученные уведомлениями:

Pest      PHPUnit

```
<?php
```

```
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;

test('orders can be shipped', function () {
    Notification::fake();

    // Выполняем доставку заказа...

    // Утверждаем, что уведомления не были отправлены...
    Notification::assertNothingSent();

    // Утверждаем, что уведомление было отправлено указанным пользователям...
    Notification::assertSentTo(
        [$user], OrderShipped::class
    );

    // Утверждаем, что уведомление не было отправлено...
    Notification::assertNotSentTo(
        [$user], AnotherNotification::class
    );

    // Утверждаем, что было отправлено заданное количество уведомлений...
    Notification::assertCount(3);
});
```

Вы можете передать замыкание в методы `assertSentTo` или `assertNotSentTo`, чтобы проверить, что было отправлено уведомление, которое проходит заданный “тест истины”. Если хотя бы одно уведомление было отправлено и прошло заданный тест, то утверждение будет успешным:

```
Notification::assertSentTo(
    $user,
    function (OrderShipped $notification, array $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);
```

## Уведомления по требованию

Если код, который вы тестируете, отправляет [уведомления по требованию](#), вы можете проверить, что уведомление по требованию было отправлено с помощью метода `assertSentOnDemand`:

```
Notification::assertSentOnDemand(OrderShipped::class);
```

Передав замыкание вторым аргументом метода `assertSentOnDemand`, вы можете определить, отправлено ли уведомление по требованию на правильный "маршрут":

```
Notification::assertSentOnDemand(
    OrderShipped::class,
    function (OrderShipped $notification, array $channels, object $notifiable) use (
        return $notifiable->routes['mail'] === $user->email;
    )
);
```

## # События уведомления

### Событие отправки уведомления

При отправке уведомления, система уведомлений запускает событие `Illuminate\Notifications\Events\NotificationSending`. Он содержит «уведомляемый» объект и сам экземпляр уведомления. Вы можете создать [прослушиватели событий](#) для этого события в своем приложении:

```
use Illuminate\Notifications\Events\NotificationSending;

class CheckNotificationStatus
{
    /**
     * Handle the given event.
     */
    public function handle(NotificationSending $event): void
    {
        // ...
    }
}
```

Уведомление не будет отправлено, если прослушиватель событий `NotificationSending` возвращает `false` из своего метода `handle`:

```
/**
 * Обработка данного события.
```

```
 */
public function handle(NotificationSending $event): bool
{
    return false;
}
```

В слушателе событий вы можете получить доступ к свойствам `notifiable`, `notification` и `channel` события, чтобы узнать больше о получателе уведомления или самом уведомлении:

```
/**
 * Обработка данного события.
 */
public function handle(NotificationSending $event): void
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

## Событие после отправки уведомления

Когда уведомление отправлено, система уведомлений запускает [событие Illuminate\Notifications\Events\NotificationSent](#). Событие содержит уведомляемую сущность и сам экземпляр уведомления. Вы можете создать [прослушиватели событий](#) для этого события в своем приложении:

```
use Illuminate\Notifications\Events\NotificationSent;

class LogNotification
{
    /**
     * Обработать переданное событие.
     */
    public function handle(NotificationSent $event): void
    {
        // ...
    }
}
```

В слушателе события вы можете получить доступ к свойствам `notifiable`, `notification`, `channel` и `response` события, чтобы узнать больше о получателе

уведомления или самом уведомлении:

```
/**
 * Обработать переданное событие.
 */
public function handle(NotificationSent $event): void
{
    // $event->channel
    // $event->notifiable
    // $event->notification
    // $event->response
}
```

## # Пользовательские каналы уведомлений

Laravel предлагает несколько каналов уведомлений, но вы можете написать свои собственные драйверы для доставки уведомлений по другим каналам. С Laravel это сделать просто. Для начала определите класс, содержащий метод `send`. Этот метод должен получать два аргумента: `$notifiable` и `$notification`.

В методе `send` вы можете вызывать методы уведомления, чтобы получить объект сообщения, понятный вашему каналу, а затем отправить уведомление необходимому экземпляру `$notifiable`:

```
<?php

namespace App\Notifications;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * Отправить переданное уведомление.
     */
    public function send(object $notifiable, Notification $notification): void
    {
        $message = $notification->toVoice($notifiable);

        // Отправка уведомления экземпляру `'$notifiable'` ...
    }
}
```

Как только ваш класс канала уведомления был определен, вы можете вернуть имя класса из метода `via` любого из ваших уведомлений. В этом примере метод вашего уведомления `toVoice` может возвращать любой объект для формирования голосовых сообщений. Например, вы можете определить свой собственный класс `VoiceMessage` для формирования таких сообщений:

```
<?php
```

```
namespace App\Notifications;

use App\Notifications\Messages\VoiceMessage;
use App\Notifications\VoiceChannel;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Получить каналы доставки уведомлений.
     */
    public function via(object $notifiable): string
    {
        return VoiceChannel::class;
    }

    /**
     * Получить содержимое голосового сообщения.
     */
    public function toVoice(object $notifiable): VoiceMessage
    {
        // ...
    }
}
```

# Разработка пакетов

## # Введение

# Примечание о фасадах

## # Обнаружение пакетов

## # Поставщики служб

## # Ресурсы

# Конфигурация

# Маршруты

# Миграции

# Языковые файлы (Переводы)

# Шаблоны

# Компоненты шаблонов

## # Команды

# Команды оптимизации

## # Публичные ресурсы

## # Публикация групп файлов

## # Введение

Пакеты – это основной способ добавления функциональности в Laravel. Пакеты могут быть чем угодно, начиная от отличной библиотеки по работе с датами, такой как [Carbon](#) или даже пакетом, который позволяет вам прикреплять файлы к моделям Eloquent, например, [Laravel Media Library](#) от Spatie.

Есть разные типы пакетов. Некоторые пакеты являются автономными, что означает, что они работают с любым фреймворком PHP. Carbon и Pest – это примеры автономных пакетов. Любой из этих пакетов можно использовать с Laravel, указав их в вашем файле [composer.json](#).

С другой стороны, некоторые пакеты специально предназначены для использования с Laravel. Эти пакеты могут иметь маршруты, контроллеры,

шаблоны и конфигурацию, специально предназначенные для улучшения приложения Laravel. Это руководство в первую очередь касается разработки пакетов для Laravel.

## Примечание о фасадах

При написании приложения Laravel, не имеет значения, используете ли вы контракты или фасады, поскольку оба подхода обеспечивают по существу равные уровни тестируемости. Однако, при написании пакетов, ваш пакет обычно не будет иметь доступа ко всем помощникам тестирования Laravel. Если вы хотите иметь возможность писать тесты для пакета, как если бы пакет был установлен внутри типичного приложения Laravel, то вы можете использовать пакет [Orchestral Testbench](#).

## # Обнаружение пакетов

Файл `bootstrap/providers.php` содержит список сервис-провайдеров, которые должны быть загружены Laravel. Однако вместо того, чтобы требовать от пользователей вручную добавлять ваш сервис-провайдер в этот список, вы можете определить провайдер в разделе `extra` файла `composer.json` вашего пакета, чтобы он автоматически загружался Laravel. Помимо сервис-провайдеров, вы также можете перечислить любые [фасады](#), которые вы хотели бы зарегистрировать:

```
"extra": {  
    "laravel": {  
        "providers": [  
            "Barryvdh\\Debugbar\\ServiceProvider"  
        ],  
        "aliases": {  
            "Debugbar": "Barryvdh\\Debugbar\\Facade"  
        }  
    },  
},
```

После того как ваш пакет будет настроен для обнаружения, Laravel автоматически зарегистрирует поставщиков и фасады пакета при его установке, создав удобство установки для пользователей вашего пакета.

## Отказ от обнаружения пакетов

Если вы являетесь пользователем пакета и хотите отключить обнаружение какого-то конкретного пакета, то вы можете указать его название в разделе `extra` файла `composer.json` вашего приложения:

```
"extra": {  
    "laravel": {  
        "dont-discover": [  
            "barryvdh/laravel-debugbar"  
        ]  
    }  
},
```

Вы можете отключить обнаружение для всех пакетов, используя метасимвол `*` внутри директивы `dont-discover` вашего приложения:

```
"extra": {  
    "laravel": {  
        "dont-discover": [  
            "*"  
        ]  
    }  
},
```

## # Поставщики служб

[Поставщики служб](#) – это точка соприкосновения между вашим пакетом и Laravel. Поставщик службы отвечает за связывание объектов в [контейнере служб](#) и информирует Laravel куда загружать ресурсы пакета, такие как шаблоны, файлы конфигурации и языковых файлов.

Поставщик службы расширяет класс `Illuminate\Support\ServiceProvider` и содержит два метода: `register` и `boot`. Базовый класс `ServiceProvider` находится в пакете `illuminate/support` Composer, который вы должны добавить в зависимости вашего собственного пакета. Чтобы узнать больше о структуре и назначении поставщиков служб, ознакомьтесь с [их документацией](#).

## # Ресурсы

# Конфигурация

Обычно, вам нужно опубликовать конфигурационный файл вашего пакета в каталог `config` приложения. Это позволит пользователям вашего пакета легко переопределить параметры конфигурации по умолчанию. Чтобы разрешить публикацию ваших файлов конфигурации, вызовите метод `publishes` в методе `boot` вашего поставщика службы:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'../../../config/courier.php' => config_path('courier.php'),
    ]);
}
```

Теперь, когда пользователи вашего пакета выполнят команду `vendor:publish` Artisan, ваш файл будет скопирован в указанное место публикации. После публикации вашей конфигурации, к ее значениям можно будет получить доступ, как к любому другому файлу конфигурации:

```
$value = config('courier.option');
```

Вы не должны определять замыкания в своих конфигурационных файлах. Они не могут быть корректно сериализованы, когда пользователи выполняют команду `config:cache` Artisan.

## Конфигурация пакета по умолчанию

Вы также можете объединить свой собственный конфигурационный файл пакета с опубликованной копией приложения. Это позволит вашим пользователям определять только те параметры, которые они действительно хотят переопределить в опубликованной копии файла конфигурации. Чтобы объединить

значения файла конфигурации, используйте метод `mergeConfigFrom` в методе `register` вашего поставщика службы.

Метод `mergeConfigFrom` принимает путь к конфигурационному файлу вашего пакета в качестве первого аргумента и имя копии конфигурационного файла приложения в качестве второго аргумента:

```
/**
 * Регистрация любых служб пакета.
 */
public function register(): void
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```

Этот метод объединяет только первый уровень массива конфигурации. Если ваши пользователи частично определяют многомерный массив конфигурации, то отсутствующие параметры не будут объединены.

## Маршруты

Если ваш пакет содержит маршруты, то вы можете загрузить их с помощью метода `loadRoutesFrom`. Этот метод автоматически определяет, закешированы ли маршруты приложения, и не загружает ваш файл маршрутов, если маршруты уже были кешированы:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->loadRoutesFrom(__DIR__.'../routes/web.php');
}
```

# Миграции

Если ваш пакет содержит [миграции базы данных](#), вы можете использовать метод `publishesMigrations`, чтобы сообщить Laravel, что указанный каталог или файл содержит миграции. Когда Laravel публикует миграции, он автоматически обновляет временные метки в их имени файла, отражая текущую дату и время:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->publishesMigrations([
        __DIR__.'/../database/migrations' => database_path('migrations'),
    ]);
}
```

# Языковые файлы (Переводы)

Если ваш пакет содержит [языковые файлы](#), то вы можете использовать метод `loadTranslationsFrom`, чтобы сообщить Laravel, как их загрузить. Например, если ваш пакет называется `courier`, то вы должны добавить следующее в метод `boot` вашего поставщика:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../resources/lang', 'courier');
}
```

Для ссылок на переводы пакетов используется синтаксическое соглашение `package::file.line`. Итак, вы можете загрузить строку приветствия пакета `courier` из файла `messages` следующим образом:

```
echo trans('courier::messages.welcome');
```

Вы можете зарегистрировать файлы перевода вашего пакета в формате JSON с помощью метода `loadJsonTranslationsFrom`. Метод принимает путь к директории,

содержащей файлы перевода вашего пакета в формате JSON:

```
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void  
{  
    $this->loadJsonTranslationsFrom(__DIR__.'/../lang');  
}
```

## Публикация языковых файлов

Если вы хотите опубликовать языковые файлы вашего пакета в каталоге `resources/lang/vendor` приложения, то вы можете использовать метод `publishes` поставщика службы. Метод `publishes` принимает массив путей пакета и желаемых мест их публикации. Например, чтобы опубликовать файлы перевода пакета `courier`, вы можете сделать следующее:

```
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void  
{  
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');  
  
    $this->publishes([  
        __DIR__.'/../resources/lang' => resource_path('lang/vendor/courier'),  
    ]);  
}
```

Теперь, когда пользователи вашего пакета выполняют команду `vendor:publish` Artisan, переводы вашего пакета будут опубликованы в указанном месте публикации.

## Шаблоны

Чтобы зарегистрировать [шаблоны](#) вашего пакета, вам необходимо указать Laravel, где они расположены. Вы можете сделать это, используя метод `loadViewsFrom` поставщика службы. Метод `loadViewsFrom` принимает два аргумента: путь к вашим шаблонам и имя вашего пакета. Например, если имя вашего пакета – `courier`, то вы должны добавить следующее в метод `boot` вашего поставщика:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

Для ссылок на шаблоны пакетов используется синтаксическое соглашение `package::view`. Итак, как только путь вашего шаблона зарегистрирован в поставщике службы, вы можете загрузить шаблон `dashboard` пакета `courier` следующим образом:

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

## Переопределение шаблонов пакета

Когда вы используете метод `loadViewsFrom`, Laravel фактически регистрирует два местоположения ваших шаблонов: каталог `resources/views/vendor` приложения и указанный вами каталог. Итак, используя пакет `courier` в качестве примера, Laravel сначала проверит, была ли размещена разработчиком пользовательская версия шаблона в каталоге `resources/views/vendor/courier`. Затем, если шаблон не был переопределен, то Laravel будет искать каталог шаблона пакета, который вы указали при вызове `loadViewsFrom`. Это позволяет пользователям пакета легко настраивать / переопределять только необходимые им шаблоны вашего пакета.

## Публикация шаблонов

Если вы хотите сделать свои шаблоны доступными для публикации в директории `resources/views/vendor` приложения, то вы можете использовать метод `publishes` поставщика службы. Метод `publishes` принимает массив, состоящий из пути к шаблону и желаемого места публикации:

Если вы хотите сделать свои шаблоны доступными для публикации в каталог `resources/views/vendor` приложения, то вы можете использовать метод `publishes` поставщика. Метод `publishes` принимает массив путей шаблонов пакета и их желаемых мест публикации:

```
/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

    $this->publishes([
        __DIR__.'/../resources/views' => resource_path('views/vendor/courier'),
    ]);
}
```

Теперь, когда пользователи вашего пакета выполняют команду `vendor:publish` Artisan, шаблоны пакета будут скопированы в указанное место публикации.

## Компоненты шаблонов

Если вы создаете пакет, который использует Blade-компоненты или размещает их в нестандартных каталогах, вам потребуется вручную зарегистрировать класс вашего компонента и его псевдоним HTML-тега, чтобы Laravel знал, где найти компонент. Обычно вы регистрируете ваши компоненты в методе `boot` сервис-провайдера вашего пакета:

Если ваш пакет содержит [компоненты шаблонов](#), то вы можете использовать метод `loadViewComponentsAs`, чтобы сообщить Laravel, как их загрузить. Метод `loadViewComponentsAs` принимает два аргумента: префикс тега компонентов и массив имен классов компонентов. Например, если префикс вашего пакета `courier` и у вас есть компоненты `Alert` и `Button`, то вы должны добавить следующее в метод `boot` вашего поставщика:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
{
    Blade::component('package-alert', AlertComponent::class);
}
```

После того как ваш компонент был зарегистрирован, его можно отобразить, используя его псевдоним тега:

```
<x-courier-alert />
```

```
<x-courier-button />
```

## Автозагрузка компонентов

В качестве альтернативы, вы можете использовать метод `componentNamespace` для автоматической загрузки классов компонентов по соглашению. Например, пакет `Nightshade` может иметь компоненты `Calendar` и `ColorPicker`, которые находятся в пространстве имен `Nightshade\Views\Components`:

```
use Illuminate\Support\Facades\Blade;

/**
 * Инициализируйте сервисы вашего пакета.
 */
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Это позволит использовать компоненты пакета с помощью синтаксиса `package-name::` их вендорного пространства имен:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade автоматически определит класс, связанный с этим компонентом, используя паскаль-кейс название компонента. Поддерживается также использование подкаталогов с помощью "точечной" нотации.

## Анонимные компоненты

Если ваш пакет содержит анонимные компоненты, то они должны быть помещены в каталог `components` каталога «views» вашего пакета (как указано в `loadViewsFrom`).

Затем вы можете отобразить их, добавив к имени компонента префикс пространства имен шаблонов пакета:

```
<x-courier::alert />
```

## Информация о пакете в Artisan

Встроенная в Laravel команда Artisan “about” предоставляет краткое описание окружения и конфигурации приложения. Пакеты могут добавлять дополнительную информацию в вывод этой команды с помощью класса [AboutCommand](#). Обычно такая информация может быть добавлена из метода `boot` сервис-провайдера вашего пакета:

```
use Illuminate\Foundation\Console\AboutCommand;

/**
 * Инициализируйте сервисы вашего пакета.
 */
public function boot(): void
{
    AboutCommand::add('Мой Пакет', fn () => ['Версия' => '1.0.0']);
}
```

Это позволит вашему пакету добавить информацию о версии и другие данные к выводу команды “about”. В данном примере, “Мой Пакет” будет отображаться в списке пакетов, и его версия будет указана как “1.0.0”.

## # Команды

Чтобы зарегистрировать команды Artisan вашего пакета в Laravel, вы можете использовать метод `commands`. Этот метод ожидает массив имен классов команд. После регистрации команд вы можете выполнять их с помощью [Artisan CLI](#):

```
use Courier\Console\Commands\InstallCommand;
use Courier\Console\Commands\NetworkCommand;

/**
 * Загрузка любых служб пакета.
 */
public function boot(): void
```

```
{  
    if ($this->app->runningInConsole()) {  
        $this->commands([  
            InstallCommand::class,  
            NetworkCommand::class,  
        ]);  
    }  
}
```

## Команды оптимизации

Команда Laravel `optimize` кэширует конфигурацию приложения, события, маршруты и представления. Используя метод `optimizes`, вы можете зарегистрировать собственные команды Artisan вашего пакета, которые должны вызываться при выполнении команд `optimize` и `optimize:clear`:

```
/**  
 * Bootstrap any package services.  
 */  
public function boot(): void  
{  
    if ($this->app->runningInConsole()) {  
        $this->optimizes(  
            optimize: 'package:optimize',  
            clear: 'package:clear-optimizations',  
        );  
    }  
}
```

## # Публичные ресурсы

В вашем пакете могут быть такие ресурсы, как изображения и скомпилированные JavaScript, CSS. Чтобы опубликовать эти ресурсы в публичном каталоге приложения, используйте метод `publishes` поставщика. В этом примере мы также добавим тег `public` группе ресурсов, который можно использовать для простой публикации групп связанных ресурсов:

```
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void
```

```
{  
    $this->publishes([  
        __DIR__.'/../public' => public_path('vendor/courier'),  
    ], 'public');  
}
```

Теперь, когда пользователи вашего пакета выполнят команду `vendor:publish`, ваши ресурсы будут скопированы в указанное место публикации. Поскольку пользователям обычно требуется перезаписывать ресурсы каждый раз при обновлении пакета, вы можете использовать флаг `--force`:

```
php artisan vendor:publish --tag=public --force
```

## # Публикация групп файлов

Вы можете публиковать файлы пакета отдельно. Например, вы можете разрешить своим пользователям публиковать конфигурационные файлы вашего пакета без необходимости публиковать остальные ресурсы вашего пакета. Вы можете сделать это, «пометив» их при вызове метода `publishes` поставщика. Например, давайте используем теги для определения двух групп публикации для пакета `courier` (`courier-config` и `courier-migrations`) в методе `boot` поставщика:

```
/**  
 * Загрузка любых служб пакета.  
 */  
public function boot(): void  
{  
    $this->publishes([  
        __DIR__.'/../config/package.php' => config_path('package.php')  
    ], 'courier-config');  
  
    $this->publishesMigrations([  
        __DIR__.'/../database/migrations/' => database_path('migrations')  
    ], 'courier-migrations');  
}
```

Теперь ваши пользователи могут публиковать эти группы отдельно, ссылаясь на их теги при выполнении команды `vendor:publish`:

```
php artisan vendor:publish --tag=courier-config
```

# Процессы

## # Введение

## # Вызов процессов

# Параметры процесса

# Вывод процесса

# Pipelines

## # Асинхронные процессы

# Идентификаторы процессов и сигналы

# Вывод асинхронного процесса

## # Параллельные процессы

# Именование процессов пула

# Идентификаторы и сигналы процессов пула

## # Тестирование

# Фиктивные процессы

# Фальсификация определенных процессов

# Подделка последовательности процессов

# Имитация жизненного цикла асинхронных процессов

# Доступные утверждения

# Предотвращение случайных процессов

## # Введение

Laravel предоставляет выразительное, минималистичное API вокруг [компонента Symfony Process](#), что позволяет вам удобно вызывать внешние процессы из вашего приложения Laravel. Возможности работы с процессами в Laravel сосредоточены на наиболее распространенных сценариях использования, обеспечивая отличный опыт разработчика.

## # Вызов процессов

Для вызова процесса вы можете использовать методы `run` и `start` предоставленные фасадом `Process`. Метод `run` вызовет процесс и будет ожидать завершения выполнения, в то время как метод `start` используется для асинхронного выполнения процесса. Оба подхода будут рассмотрены в этой документации. Давайте сначала изучим, как вызвать базовый синхронный процесс и проверить его результат:

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

return $result->output();
```

Конечно, экземпляр `Illuminate\Contracts\Process\ProcessResult` возвращаемый методом `run` предоставляет разнообразие полезных методов, которые можно использовать для анализа результата выполнения процесса:

```
$result = Process::run('ls -la');

$result->successful();
$result->failed();
$result->exitCode();
$result->output();
$result->errorOutput();
```

## Обработка исключений

Если у вас есть результат выполнения процесса, и вы хотите выбросить экземпляр `Illuminate\Process\Exceptions\ProcessFailedException`, если код завершения больше нуля (что указывает на ошибку), вы можете использовать методы `throw` и `throwIf`. Если процесс не завершился ошибкой, будет возвращен экземпляр результата процесса:

```
$result = Process::run('ls -la')->throw();

$result = Process::run('ls -la')->throwIf($condition);
```

## Параметры процесса

## Путь к рабочему каталогу

Вы можете использовать метод `path` для указания рабочего каталога процесса. Если этот метод не вызывается, процесс унаследует рабочий каталог текущего выполняющегося скрипта PHP:

```
$result = Process::path(__DIR__)->run('ls -la');
```

## Ввод

Вы можете предоставить ввод через "стандартный ввод" процесса, используя метод `input`:

```
$result = Process::input('Hello World')->run('cat');
```

## Таймаут

По умолчанию процессы будут выбрасывать экземпляр `Illuminate\Process\Exceptions\ProcessTimedOutException` если выполняются более 60 секунд. Однако вы можете настроить это поведение с помощью метода `timeout`:

```
$result = Process::timeout(120)->run('bash import.sh');
```

Или, если вы хотите полностью отключить таймаут процесса, вы можете вызвать метод `forever`:

```
$result = Process::forever()->run('bash import.sh');
```

Метод `idleTimeout` можно использовать для указания максимального количества секунд, в течение которых процесс может выполняться, не возвращая никакого вывода:

```
$result = Process::timeout(60)->idleTimeout(30)->run('bash import.sh');
```

## Переменные среды

Переменные среды могут быть предоставлены процессу с помощью метода `env`. Вызванный процесс также унаследует все переменные среды, определенные в вашей системе:

```
$result = Process::forever()  
    ->env(['IMPORT_PATH' => __DIR__])  
    ->run('bash import.sh');
```

Если вы хотите удалить унаследованную переменную среды из вызванного процесса, вы можете предоставить этой переменной среды значение `false`:

```
$result = Process::forever()  
    ->env(['LOAD_PATH' => false])  
    ->run('bash import.sh');
```

## Режим TTY

Метод `tty` можно использовать для включения режима TTY для вашего процесса. Режим TTY соединяет ввод и вывод процесса с вводом и выводом вашей программы, что позволяет вашему процессу открывать редактор, такой как Vim или Nano, как процесс:

```
Process::forever()->tty()->run('vim');
```

## Вывод процесса

Как уже обсуждалось ранее, вывод процесса может быть получен с использованием методов `output` (`stdout`) и `errorOutput` (`stderr`) в результате выполнения процесса:

```
use Illuminate\Support\Facades\Process;  
  
$result = Process::run('ls -la');  
  
echo $result->output();  
echo $result->errorOutput();
```

Однако вывод также можно собрать в реальном времени, передав замыкание в качестве второго аргумента методу `run`. Замыкание будет получать два аргумента: “тип” вывода (`stdout` или `stderr`) и сам вывод в виде строки:

```
$result = Process::run('ls -la', function (string $type, string $output) {
    echo $output;
});
```

Laravel также предлагает методы `seeInOutput` and `seeInErrorOutput`, которые предоставляют удобный способ определить, содержится ли заданная строка в выводе процесса:

```
if (Process::run('ls -la')->seeInOutput('laravel')) {
    // ...
}
```

## Отключение вывода процесса

Если ваш процесс записывает большое количество вывода, которое вам неинтересно, вы можете сэкономить память, полностью отключив получение вывода. Для этого вызовите метод `quietly` при создании процесса:

```
use Illuminate\Support\Facades\Process;

$result = Process::quietly()->run('bash import.sh');
```

## Pipelines

Иногда вам может потребоваться передать вывод одного процесса в качестве ввода для другого процесса. Это часто называется “перенаправлением” (piping) вывода одного процесса в другой. Метод `pipe`, предоставляемый фасадом `Process` упрощает это. Метод `pipe` выполнит связанные процессы синхронно и вернет результат последнего процесса в `pipeline`:

```
use Illuminate\Process\Pipe;
use Illuminate\Support\Facades\Process;

$result = Process::pipe(function (Pipe $pipe) {
    $pipe->command('cat example.txt');
```

```
$pipe->command('grep -i "laravel"');
});

if ($result->successful()) {
    // ...
}
```

Если вам не нужно настраивать отдельные процессы, составляющие pipeline, вы можете просто передать массив строк команд методу `pipe`:

```
$result = Process::pipe([
    'cat example.txt',
    'grep -i "laravel"',
]);
```

Вывод процесса можно собрать в реальном времени, передав замыкание в качестве второго аргумента методу `pipe` замыкание будет принимать два аргумента: "тип" вывода (`stdout` или `stderr`) и сам вывод в виде строки:

```
$result = Process::pipe(function (Pipe $pipe) {
    $pipe->command('cat example.txt');
    $pipe->command('grep -i "laravel"');
}, function (string $type, string $output) {
    echo $output;
});
```

Laravel также позволяет назначать строковые ключи каждому процессу, содержащемуся в pipeline, с помощью метода `as`. Этот ключ также будет передан в замыкание вывода, предоставленное методу `pipe`, что позволит вам определить, к какому процессу относится вывод:

```
$result = Process::pipe(function (Pipe $pipe) {
    $pipe->as('first')->command('cat example.txt');
    $pipe->as('second')->command('grep -i "laravel"');
})->start(function (string $type, string $output, string $key) {
    // ...
});
```

## # Асинхронные процессы

В то время как метод `run` вызывает процессы синхронно, метод `start` может быть использован для вызова процесса асинхронно. Это позволяет вашему приложению продолжать выполнение других задач, пока процесс выполняется в фоновом режиме. После вызова процесса вы можете использовать метод `running` для определения, выполняется ли процесс:

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    // ...
}

$result = $process->wait();
```

Как вы могли заметить, вы можете вызвать метод `wait`, чтобы дождаться завершения выполнения процесса и получить экземпляр результата процесса:

```
$process = Process::timeout(120)->start('bash import.sh');

// ...

$result = $process->wait();
```

## Идентификаторы процессов и сигналы

Метод `id` может быть использован для получения присвоенного операционной системой идентификатора выполняющегося процесса:

```
$process = Process::start('bash import.sh');

return $process->id();
```

Вы можете использовать метод `signal` для отправки “сигнала” запущенному процессу. Список предопределенных констант сигналов можно найти в [документации по PHP](#):

```
$process->signal(SIGUSR2);
```

# Вывод асинхронного процесса

Во время выполнения асинхронного процесса вы можете получить доступ к его текущему выводу с помощью методов `output` и `errorOutput`. Однако для получения вывода процесса, который произошел после последнего вывода, вы можете использовать методы `latestOutput` и `latestErrorOutput`:

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    echo $process->latestOutput();
    echo $process->latestErrorOutput();

    sleep(1);
}
```

Как и в случае с методом `run`, для асинхронных процессов вывод также можно собирать в реальном времени, передав замыкание вторым аргументом методу `start`. Замыкание будет получать два аргумента: “тип” вывода (`stdout` или `stderr`) и саму строку вывода:

```
$process = Process::start('bash import.sh', function (string $type, string $output) {
    echo $output;
});

$result = $process->wait();
```



Вместо того чтобы просто ждать завершения процесса, вы можете использовать метод `waitFor`, чтобы остановить ожидание, как только процесс выведет нужный результат. Laravel прекратит ожидание завершение процесса, когда замыкание, переданное в метод `waitFor`, вернёт `true`:

```
$process = Process::start('bash import.sh');

$process->waitFor(function (string $type, string $output) {
    return $output === 'Ready...';
});
```

## # Параллельные процессы

Laravel также делает легким управление пулом одновременных асинхронных процессов, что позволяет легко выполнять множество задач параллельно. Для начала используйте метод `pool`, который принимает замыкание, получающее экземпляр `Illuminate\Process\Pool`.

Внутри этого замыкания вы можете определить процессы, принадлежащие пулу. После запуска пула процессов с помощью метода `start` вы можете получить доступ к [коллекции](#) запущенных процессов с помощью метода `running`:

```
use Illuminate\Process\Pool;
use Illuminate\Support\Facades\Process;

$pool = Process::pool(function (Pool $pool) {
    $pool->path(__DIR__)->command('bash import-1.sh');
    $pool->path(__DIR__)->command('bash import-2.sh');
    $pool->path(__DIR__)->command('bash import-3.sh');
})->start(function (string $type, string $output, int $key) {
    // ...
});

while ($pool->running()->isNotEmpty()) {
    // ...
}

$results = $pool->wait();
```

Как видите, вы можете дождаться завершения выполнения всех процессов в пуле и получить их результаты с помощью метода `wait`. Метод `wait` возвращает объект, доступный в виде массива, который позволяет получить экземпляр результата каждого процесса в пуле по его ключу:

```
$results = $pool->wait();

echo $results[0]->output();
```

Или, для удобства, можно использовать метод `concurrently` для запуска асинхронного пула процессов и немедленного ожидания их результатов. Это может обеспечить особенно выразительный синтаксис при использовании в сочетании с возможностями деструктуризации массивов в PHP:

```
[$first, $second, $third] = Process::concurrently(function (Pool $pool) {
    $pool->path(__DIR__)->command('ls -la');
    $pool->path(app_path())->command('ls -la');
    $pool->path(storage_path())->command('ls -la');
});

echo $first->output();
```

## Именование процессов пула

Доступ к результатам пула процессов по числовому ключу не очень выразителен; поэтому Laravel позволяет вам назначать строковые ключи каждому процессу в пуле с помощью метода `as`. Этот ключ также будет передан замыканию, предоставленному методу `start`, что позволит вам определить, к какому процессу относится вывод:

```
$pool = Process::pool(function (Pool $pool) {
    $pool->as('first')->command('bash import-1.sh');
    $pool->as('second')->command('bash import-2.sh');
    $pool->as('third')->command('bash import-3.sh');
})->start(function (string $type, string $output, string $key) {
    // ...
});

$results = $pool->wait();

return $results['first']->output();
```

## Идентификаторы и сигналы процессов пула

Поскольку метод `running` пула процессов предоставляет коллекцию всех вызванных процессов внутри пула, вы легко можете получить доступ к идентификаторам процессов в основном пуле:

```
$processIds = $pool->running()->each->id();
```

И, для удобства, вы можете вызвать метод `signal` пула процессов, чтобы отправить сигнал каждому процессу внутри пула:

```
$pool->signal(SIGUSR2);
```

## # Тестирование

Многие службы Laravel предоставляют функциональность для удобного и выразительного написания тестов, и служба процессов Laravel не является исключением. Метод `fake` фасада `Process` позволяет вам указать Laravel возвращать фиктивные / заглушечные результаты при вызове процессов.

### Фиктивные процессы

Для исследования возможности фальсификации процессов в Laravel, представим маршрут, который вызывает процесс:

```
use Illuminate\Support\Facades\Process;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
    Process::run('bash import.sh');

    return 'Import complete!';
});
```

При тестировании этого маршрута мы можем указать Laravel вернуть поддельный успешный результат для каждого вызванного процесса, вызвав метод `fake` на фасаде `Process` без аргументов. Кроме того, мы даже можем [проверить](#), что определенный процесс был “запущен”:

Pest      PHPUnit

```
<?php

use Illuminate\Process\PendingProcess;
use Illuminate\Contracts\Process\ProcessResult;
use Illuminate\Support\Facades\Process;

test('process is invoked', function () {
    Process::fake();

    $response = $this->get('/import');
```

```
// Simple process assertion...
Process::assertRan('bash import.sh');

// Or, inspecting the process configuration...
Process::assertRan(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'bash import.sh' &&
        $process->timeout === 60;
});

});
```

Как обсуждалось, вызов метода `fake` фасада `Process` указывает Laravel всегда возвращать успешный результат процесса без вывода. Тем не менее, вы легко можете указать вывод и код завершения для поддельных процессов с использованием метода `result` фасада `Process`:

```
Process::fake([
    '*' => Process::result(
        output: 'Test output',
        errorOutput: 'Test error output',
        exitCode: 1,
    ),
]);
```

## Фальсификация определенных процессов

Как вы могли заметить в предыдущем примере, фасад `Process` позволяет вам указывать различные поддельные результаты для каждого процесса, передав массив методу `fake`.

Ключи массива должны представлять шаблоны команд, которые вы хотите подделать, и их результаты. Символ `*` может быть использован в качестве символа-заменителя. Любые команды процессов, которые не были подделаны, будут действительно вызваны. Вы можете использовать метод `result` фасада `Process` для создания заглушек / фейковых результатов для этих команд:

```
Process::fake([
    'cat *' => Process::result(
        output: 'Test "cat" output',
    ),
    'ls *' => Process::result(
        output: 'Test "ls" output',
```

```
),
]);
```

Если вам не нужно настраивать код завершения или вывод ошибок поддельного процесса, вам может быть удобнее указывать результаты фейкового процесса в виде простых строк:

```
Process::fake([
    'cat *' => 'Test "cat" output',
    'ls *' => 'Test "ls" output',
]);
```

## Подделка последовательности процессов

Если код, который вы тестируете, вызывает несколько процессов с одной и той же командой, вы можете назначить различные фейковые результаты каждому вызову процесса. Вы можете сделать это с помощью метода `sequence` фасада `Process`:

```
Process::fake([
    'ls *' => Process::sequence()
        ->push(Process::result('First invocation'))
        ->push(Process::result('Second invocation')),
]);
```

## Имитация жизненного цикла асинхронных процессов

До сих пор мы в основном говорили о фейковых процессах, которые вызываются синхронно с использованием метода `run`. Однако, если вы пытаетесь протестировать код, который взаимодействует с асинхронными процессами, вызываемыми с помощью `start`, вам может потребоваться более сложный подход к описанию ваших фейковых процессов.

Например, представим следующий маршрут, который взаимодействует с асинхронным процессом:

```
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
```

```

$process = Process::start('bash import.sh');

while ($process->running()) {
    Log::info($process->latestOutput());
    Log::info($process->latestErrorOutput());
}

return 'Done';
});

```

To properly fake this process, we need to be able to describe how many times the `running` method should return `true`. In addition, we may want to specify multiple lines of output that should be returned in sequence. To accomplish this, we can use the `Process` facade's `describe` method:

```

Process::fake([
    'bash import.sh' => Process::describe()
        ->output('First line of standard output')
        ->errorOutput('First line of error output')
        ->output('Second line of standard output')
        ->exitCode(0)
        ->iterations(3),
]);

```

Чтобы корректно подделать этот процесс, нам нужно иметь возможность описать, сколько раз метод `running` должен возвращать `true`. Кроме того, по желанию, мы можем указать несколько строк вывода, которые должны быть возвращены последовательно. Для этого мы можем использовать метод `describe` фасада `Process`:

## Доступные утверждения

Как [уже обсуждалось ранее](#), Laravel предоставляет несколько утверждений процессов для ваших функциональных тестов. Рассмотрим каждое из этих утверждений.

### `assertRan`

Утверждение, что определенный процесс был вызван:

```
use Illuminate\Support\Facades\Process;
```

```
Process::assertRan('ls -la');
```

Метод `assertRan` также принимает замыкание, которое получит экземпляр процесса и результат процесса, что позволяет вам проверить настроенные опции процесса. Если это замыкание возвращает `true`, утверждение будет “пройдено”:

```
Process::assertRan(fn ($process, $result) =>
    $process->command === 'ls -la' &&
    $process->path === __DIR__ &&
    $process->timeout === 60
);
```

Переменная `$process`, переданная в замыкание `assertRan`, является экземпляром `Illuminate\Process\PendingProcess`, в то время как `$result` – экземпляром `Illuminate\Contracts\Process\ProcessResult`.

## assertDidntRun

Утверждение, что определенный процесс не был вызван:

```
use Illuminate\Support\Facades\Process;

Process::assertDidntRun('ls -la');
```

Как и метод `assertRan`, метод `assertDidntRun` также принимает замыкание, которое получит экземпляр процесса и результат процесса, что позволяет вам проверить настроенные опции процесса. Если это замыкание возвращает `true`, утверждение будет “провалено”:

```
Process::assertDidntRun(fn (PendingProcess $process, ProcessResult $result) =>
    $process->command === 'ls -la'
);
```

## assertRanTimes

Утверждение, что определенный процесс был вызван определенное количество раз:

```
use Illuminate\Support\Facades\Process;

Process::assertRanTimes('ls -la', times: 3);
```

Метод `assertRanTimes` также принимает замыкание, которое получит экземпляр процесса и результат процесса, что позволяет вам проверить настроенные опции процесса. Если это замыкание возвращает `true` и процесс был вызван указанное количество раз, утверждение будет “пройдено”:

```
Process::assertRanTimes(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'ls -la';
}, times: 3);
```

## Предотвращение случайных процессов

Если вы хотите убедиться, что все вызванные процессы были подделаны в пределах отдельного теста или набора тестов, вы можете вызвать метод `preventStrayProcesses`. После вызова этого метода любые процессы, для которых нет соответствующего поддельного результата, вызовут исключение, а не фактический процесс:

```
use Illuminate\Support\Facades\Process;

Process::preventStrayProcesses();

Process::fake([
    'ls *' => 'Test output...',
]);

// Fake response is returned...
Process::run('ls -la');

// An exception is thrown...
Process::run('bash import.sh');
```

# Очереди

## # Введение

- # Соединения и очереди
- # Предварительная подготовка драйверов

## # Создание заданий

- # Генерация класса задания
- # Структура класса задания
- # Уникальные задания
- # Шифрование заданий

## # Посредник (*middleware*) задания

- # Ограничение частоты
- # Предотвращение дублирования задания
- # Ограничение частоты генерации исключений
- # Пропуск заданий

## # Отправка заданий

- # Отложенная отправка
- # Синхронная отправка
- # Задания и транзакции базы данных
- # Цепочка заданий
- # Настройка соединения и очереди
- # Указание максимального количества попыток задания / значений тайм-аута
- # Обработка ошибок

## # Пакетная обработка заданий

- # Определение пакета заданий
- # Отправка пакета заданий
- # Добавление заданий в пакет заданий
- # Инспектирование пакета
- # Отмена пакетов
- # Отказы в пакете заданий
- # Очистка пакетов

- # Хранение пакетов в DynamoDB
- # Анонимные очереди
- # Запуск обработчика очереди
  - # Команда queue:work
  - # Приоритеты очереди
  - # Обработчики очереди и развертывание
  - # Истечение срока и тайм-ауты задания
- # Конфигурация Supervisor
- # Разбор неудачных заданий
  - # Очистка после неудачных заданий
  - # Повторная попытка выполнения неудачных заданий
  - # Игнорирование отсутствующих моделей
  - # Удаление неудачных заданий
  - # Хранение неудачных заданий в DynamoDB
  - # Отключение хранилища неудачных заданий
  - # События неудачных заданий
- # Удаление заданий из очередей
- # Мониторинг очередей
- # Тестирование
  - # Подделка определённого списка заданий
  - # Тестирование цепочку заданий
  - # Тестирование пакетов заданий
  - # Тестирование взаимодействия заданий и очередей
- # События заданий

## # Введение

При создании веб-приложения у вас могут быть некоторые задачи, такие как синтаксический анализ и сохранение загруженного файла CSV, выполнение которых во время обычного веб-запроса занимает слишком много времени. К счастью, Laravel позволяет легко создавать задания (*jobs*) в очереди (*queue*), которые могут обрабатываться в фоновом режиме. Перемещая трудоемкие задания в очередь и выполняя их в фоне, ваше приложение может быстрее обрабатывать веб-запросы и быстрее отвечать клиенту.

Очереди Laravel предоставляют унифицированный API для различных серверных служб очередей, таких как [Amazon SQS](#), [Redis](#) или даже обычная реляционная база данных.

Параметры конфигурации очереди Laravel хранятся в файле конфигурации вашего приложения `config/queue.php`. В этом файле вы найдете конфигурации подключения для каждого из драйверов очереди фреймворка: база данных, [Amazon SQS](#), [Redis](#) и [Beanstalkd](#), а также синхронный драйвер для немедленного выполнения задания (используется во время локальной разработки). Также имеется драйвер очереди `null`, который просто выбрасывает задания из очереди, не исполняя их.

Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full [Horizon documentation](#) for more information.

## Соединения и очереди

Прежде чем приступить к работе с очередями Laravel, важно понять различие между «соединениями» и «очередями». В конфигурационном файле `config/queue.php` есть массив `connections`. Этот параметр определяет подключения к серверным службам очередей, таким как Amazon SQS, Beanstalk или Redis. Однако любое указанное «соединение» очереди может иметь несколько «очередей», которые можно рассматривать как разные стеки или пачки поочередных заданий.

Обратите внимание, что каждый пример конфигурации соединения в файле конфигурации `queue` содержит ключ `queue`. Это очередь по умолчанию, в которую будут отправляться задания при их отправке в определенное соединение. Другими словами, если вы отправляете задание без явного определения очереди, в которую оно должно быть отправлено, задание будет поставлено в очередь, определённую в ключе `queue` конфигурации соединения:

```
use App\Jobs\ProcessPodcast;  
  
// Это задание отправляется в очередь `default` соединения по умолчанию ...  
ProcessPodcast::dispatch();
```

```
// Это задание отправляется в очередь `emails` соединения по умолчанию ...
ProcessPodcast::dispatch()->onQueue('emails');
```

Некоторым приложениям может не понадобиться помещать задания в несколько очередей, вместо этого предпочитая иметь одну простую очередь. Однако отправка заданий в несколько очередей может быть особенно полезна для приложений, определяющих приоритеты или сегментацию процесса обработки заданий, поскольку обработчик очереди Laravel позволяет вам указать, какие очереди он должен обрабатывать по приоритету. Например, если вы помещаете задания в очередь `high`, то вы можете запустить обработчик, который даст им более высокий приоритет обработки:

```
php artisan queue:work --queue=high,default
```

## Предварительная подготовка драйверов

### База данных

Чтобы использовать драйвер очереди `database`, вам понадобится таблица базы данных для хранения заданий. Обычно это включено в стандартный файл Laravel `0001_01_01_000002_create_jobs_table.php` `databasemigration`; однако, если ваше приложение не содержит этой миграции, вы можете использовать Artisan-команду `make:queue-table` для ее создания:

```
php artisan make:queue-table
```

```
php artisan migrate
```

### Redis

Чтобы использовать драйвер очереди `redis`, вы должны настроить соединение с базой данных Redis в файле конфигурации `config/database.php`.

Возможности Redis `serializer` и `compression` не поддерживаются драйвером очереди `redis`.

## Кластер Redis

Если ваше соединение с очередью Redis использует кластер Redis, то имена ваших очередей должны содержать [ключевой хеш-тег](#). Это необходимо для того, чтобы все ключи Redis для указанной очереди были поставлены в один и тот же хеш-слот:

```
'redis' => [
    'driver' => 'redis',
    'connection' => env('REDIS_QUEUE_CONNECTION', 'default'),
    'queue' => env('REDIS_QUEUE', '{default}'),
    'retry_after' => env('REDIS_QUEUE_RETRY_AFTER', 90),
    'block_for' => null,
    'after_commit' => false,
],
```

## Блокировка

При использовании очереди Redis вы можете использовать параметр конфигурации `block_for`, чтобы указать, как долго драйвер должен ждать, пока задание станет доступным, прежде чем выполнить итерацию через рабочий цикл и повторно опросить базу данных Redis.

Настройка этого значения зависит от загрузки очереди и может быть более эффективной, чем постоянный опрос базы данных Redis на предмет новых заданий. Например, вы можете установить значение `5`, чтобы указать, что драйвер должен блокироваться на пять секунд, ожидая, пока задание станет доступным:

```
'redis' => [
    'driver' => 'redis',
    'connection' => env('REDIS_QUEUE_CONNECTION', 'default'),
    'queue' => env('REDIS_QUEUE', 'default'),
    'retry_after' => env('REDIS_QUEUE_RETRY_AFTER', 90),
    'block_for' => 5,
    'after_commit' => false,
],
```

Установка для `block_for` значения `0` заставит обработчиков очереди блокироваться на неопределенный срок, пока задание не станет доступным. Это также предотвратит обработку

таких сигналов, как `SIGTERM`, до тех пор, пока не будет обработано следующее задание.

## Дополнительные зависимости драйверов

Для перечисленных драйверов очереди необходимы следующие зависимости. Эти зависимости могут быть установлены через менеджер пакетов Composer:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~5.0`
- Redis: `predis/predis ~2.0` or `phpredis` PHP extension

## # Создание заданий

### Генерация класса задания

Чтобы сгенерировать новое задание, используйте команду `make:job Artisan`. Эта команда поместит новый класс задания в каталог `app/Jobs` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его:

```
php artisan make:job ProcessPodcast
```

Сгенерированный класс будет реализовывать интерфейс `Illuminate\Contracts\Queue\ShouldQueue`, указывая Laravel, что задание должно быть поставлено в очередь для асинхронного выполнения.

Заготовки (stub) заданий можно настроить с помощью [публикации заготовок](#).

### Структура класса задания

Классы заданий очень простые, обычно они содержат только метод `handle`, который вызывается, когда задание обрабатывается очередью. Для начала рассмотрим пример класса задания. В этом примере мы представим, что управляем службой публикации подкастов и нам необходимо обработать загруженные файлы подкастов перед их публикацией:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;

class ProcessPodcast implements ShouldQueue
{
    use Queueable;

    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * Выполнить задание.
     */
    public function handle(AudioProcessor $processor): void
    {
        // Обработка загруженного подкаста ...
    }
}
```

Обратите внимание, что в этом примере мы смогли передать [модель Eloquent](#) непосредственно в конструктор задания. Благодаря трейту `Queueable`, который использует задание, модели Eloquent и их загруженные отношения будут корректно сериализованы и десериализованы при обработке задания.

Если ваше задание в очереди принимает модель Eloquent в своем конструкторе, в очередь будет сериализован только идентификатор модели. Когда задание действительно обрабатывается, система очередей автоматически повторно извлекает полный экземпляр модели и его загруженные отношения из базы данных. Такой подход к сериализации модели позволяет отправлять в драйвер очереди гораздо меньший объем данных.

## Внедрение зависимости метода handle

Метод `handle` вызывается, когда задание обрабатывается очередью. Обратите внимание, что мы можем объявить тип зависимости в методе `handle` задания.

[Контейнер служб](#) Laravel автоматически внедряет эти зависимости.

Если вы хотите получить полный контроль над тем, как контейнер внедряет зависимости в метод `handle`, вы можете использовать метод `bindMethod` контейнера. Метод `bindMethod` принимает функцию, которая получает задание и контейнер. В функции вы можете вызывать метод `handle`. Обычно вы должны вызывать `bindMethod` из метода `boot` вашего [сервис-провайдера App\Providers\AppServiceProvider](#):

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;
use Illuminate\Contracts\Foundation\Application;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function (ProcessPodcast $job) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

Бинарные данные, например, необработанное содержимое изображения, должны быть переданы через функцию `base64_encode` перед передачей заданию. В противном случае задание может неправильно сериализоваться в JSON при отправке в очередь.

## Очередь отношений

Поскольку все загруженные отношения модели Eloquent также сериализуются при постановке задания в очередь, сериализованная строка задания иногда может стать довольно объемной. Более того, когда задача десериализуется, и отношения модели повторно извлекаются из базы данных, они будут извлечены в полном объеме. Любые предыдущие ограничения отношений, которые были применены до того, как модель была сериализована в процессе постановки задания в очередь, не будут применены, когда задача будет десериализована. Поэтому, если вам

необходимо работать с подмножеством определенного отношения, вам следует повторно наложить ограничение на это отношение внутри вашей задачи в очереди.

Или, чтобы предотвратить сериализацию отношений, вы можете вызвать метод модели `withoutRelations` при установке значения свойства в задание. Этот метод вернет экземпляр модели без загруженных связей:

```
/**
 * Создать новый экземпляр задания.
 */
public function __construct(
    Podcast $podcast,
) {
    $this->podcast = $podcast->withoutRelations();
}
```

Если вы используете свойства конструктора PHP и хотите указать, что модель Eloquent не должна сериализовать свои отношения, вы можете использовать атрибут `withoutRelations`:

```
use Illuminate\Queue\Attributes\WithoutRelations;

/**
 * Create a new job instance.
 */
public function __construct(
    #[WithoutRelations]
    public Podcast $podcast,
) {}
```

Если задание получает коллекцию или массив моделей Eloquent вместо одной модели, отношения между моделями в этой коллекции не будут восстановлены при десериализации и выполнении задания. Это необходимо для предотвращения чрезмерного использования ресурсов в заданиях, связанных с большим количеством моделей.

## Уникальные задания

Для уникальных заданий требуется драйвер кеша, поддерживающий [блокировки](#). В настоящее время драйверы кеширования `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` поддерживают атомарные блокировки. Кроме того, уникальность заданий не учитывается при пакетной обработке.

Иногда требуется убедиться, что только один экземпляр определенного задания находится в очереди в любой момент времени. Вы можете сделать это, реализовав интерфейс `ShouldBeUnique` в своем классе задания. Этот интерфейс не требует от вас определения каких-либо дополнительных методов в вашем классе:

```
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

В приведенном выше примере задание `UpdateSearchIndex` уникально. Таким образом, задание не будет отправлено, если другой экземпляр задания уже находится в очереди и еще не завершил обработку.

В некоторых случаях вам может потребоваться определить конкретный «ключ», делающий задание уникальным, или вы можете указать тайм-аут, по истечении которого задание больше не считается уникальным. Для этого вы можете определить свойства или методы `uniqueId` и `uniqueFor` в своем классе задания:

```
<?php

use App\Models\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
```

```
/**  
 * Экземпляр продукта.  
 *  
 * @var \App\Product  
 */  
public $product;  
  
/**  
 * Количество секунд, по истечении которых уникальная блокировка задания будет снята.  
 *  
 * @var int  
 */  
public $uniqueFor = 3600;  
  
/**  
 */  
public function uniqueId(): string  
{  
    return $this->product->id;  
}  
}
```

В приведенном выше примере задание `UpdateSearchIndex` уникально по идентификатору продукта. Таким образом, любые новые отправленные задания с тем же идентификатором продукта будут игнорироваться, пока существующее задание не завершит обработку. Кроме того, если существующее задание не будет обработано в течение одного часа, уникальная блокировка будет снята, и в очередь может быть отправлено другое задание с таким же уникальным ключом.

Если ваше приложение отправляет задания с нескольких веб-серверов или контейнеров, вам следует убедиться, что все ваши серверы взаимодействуют с одним и тем же сервером центрального кэша, чтобы Laravel мог точно определить, является ли задание уникальным.

## Сохранение уникальности задания только до начала обработки

По умолчанию уникальные задания «разблокируются» после того, как задание завершит обработку или потерпит неудачу во всех повторных попытках. Однако, могут возникнуть ситуации, когда вы захотите, чтобы ваше задание было разблокировано непосредственно перед его обработкой. Для этого ваше задание должно реализовать контракт `ShouldBeUniqueUntilProcessing` вместо контракта `ShouldBeUnique`:

```
<?php

use App\Models\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    // ...
}
```

## Блокировки уникальных заданий

За кулисами, когда отправляется задание `ShouldBeUnique`, Laravel пытается получить блокировку с ключом `uniqueId`. Если блокировка не получена, задание не отправляется. Эта блокировка снимается, когда задание завершает обработку или терпит неудачу во всех повторных попытках. По умолчанию Laravel будет использовать драйвер кеша, назначенный по умолчанию, для получения этой блокировки. Однако, если вы хотите использовать другой драйвер для получения блокировки, вы можете определить метод `uniqueVia`, возвращающий драйвер кеша, который следует использовать:

```
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Support\Facades\Cache;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...

    /**
     * Получить драйвер кеша для блокировки уникального задания.
     */
    public function uniqueVia(): Repository
    {
        return Cache::driver('redis');
```

```
    }  
}
```

Если вам нужно ограничить только параллельную обработку задания, используйте вместо этого посредник [WithoutOverlapping](#).

## Шифрование заданий

Laravel позволяет вам обеспечить конфиденциальность и целостность данных задания с помощью [шифрования](#). Для начала просто добавьте интерфейс [ShouldBeEncrypted](#) в класс задания. Как только этот интерфейс будет добавлен в класс, Laravel автоматически зашифрует ваше задание, прежде чем поместить его в очередь:

```
<?php  
  
use Illuminate\Contracts\Queue\ShouldBeEncrypted;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class UpdateSearchIndex implements ShouldQueue, ShouldBeEncrypted  
{  
    // ...  
}
```

## # Посредник (middleware) задания

Посредник задания позволяет обернуть пользовательскую логику вокруг выполнения заданий в очереди, уменьшая шаблонность самих заданий. Например, рассмотрим следующий метод [handle](#), который использует функции ограничения частоты, позволяющие обрабатывать только одно задание каждые пять секунд:

```
use Illuminate\Support\Facades\Redis;  
  
/**  
 * Выполнить задание.  
 */
```

```

public function handle(): void
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
        info('Lock obtained...');

        // Обработка задания ...
    }, function () {
        // Не удалось получить блокировку ...

        return $this->release(5);
    });
}

```

Хотя этот код действителен, реализация метода `handle` становится «шумной», так как она загромождена логикой ограничения частоты Redis. Кроме того, эта логика ограничения частоты должна быть продублирована для любых других заданий, для которых мы хотим установить ограничение частоты.

Вместо ограничения частоты в методе `handle` мы могли бы определить посредника задания, который обрабатывает ограничение частоты. В Laravel нет места по умолчанию для посредников заданий, поэтому вы можете разместить их в любом месте вашего приложения. В этом примере мы поместим его в каталог `app/Jobs/Middleware`:

```

<?php

namespace App\Jobs\Middleware;

use Closure;
use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * Обработать задание в очереди.
     *
     * @param \Closure(object): void $next
     */
    public function handle(object $job, Closure $next): void
    {
        Redis::throttle('key')
            ->block(0)->allow(1)->every(5)
            ->then(function () use ($job, $next) {
                // Блокировка получена ...
            });
    }
}

```

```
$next($job);
}, function () use ($job) {
    // Не удалось получить блокировку ...

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

Как вы можете видеть, как и [посредник маршрута](#), посредник задания получает обрабатываемое задание и функцию, которая должна быть вызвана для продолжения обработки задания.

После создания посредника задания он может быть назначен заданию, вернув их из метода `middleware` задания. Этот метод не существует для заданий, созданных с помощью команды `make:job` Artisan, поэтому вам нужно будет вручную добавить его в свой класс задания:

```
use App\Jobs\Middleware\RateLimited;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited];
}
>
```

Посредник заданий также может быть назначен слушателям событий, почтовым отправлениям или уведомлениям – если они выполняются через очередь.

## Ограничение частоты

Хотя мы только что продемонстрировали, как написать собственного посредника, ограничивающего частоту, Laravel на самом деле включает посредника, который вы можете использовать для задания ограничения частоты. Как и [ограничители частоты маршрута](#), ограничители частоты задания определяются с помощью метода `for` фасада `RateLimiter`.

Например, вы можете разрешить пользователям выполнять резервное копирование своих данных один раз в час, при этом не накладывая таких ограничений на премиум-клиентов. Для этого вы можете определить `RateLimiter` в методе `boot` вашего `AppServiceProvider`:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Загрузка любых служб приложения.
 *
 * @return void
 */
public function boot(): void
{
    RateLimiter::for('backups', function (object $job) {
        return $job->user->vipCustomer()
            ? Limit::none()
            : Limit::perHour(1)->by($job->user->id);
    });
}
```

В приведенном выше примере мы определили часовой лимит частоты; однако вы можете легко определить ограничение на основе минут, используя метод `perMinute`. Кроме того, вы можете передать любое значение методу `by` ограничения; однако это значение чаще всего используется для сегментации ограничений частоты по клиентам:

```
return Limit::perMinute(50)->by($job->user->id);
```

После того как вы определили ограничение частоты, вы можете назначить ограничитель частоты своему заданию резервного копирования с помощью посредника `Illuminate\Queue\Middleware\RateLimited`. Каждый раз, когда задание превышает ограничение частоты, этот посредник отправляет задание обратно в

очередь с соответствующей задержкой в зависимости от продолжительности ограничения частоты.

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited('backups')];
}
```

Возвращение задания с ограниченной частотой обратно в очередь все равно увеличит общее количество «попыток» (`attempts`) задания. Возможно, вы захотите соответствующим образом настроить свойства `tries` и `maxExceptions` в своем классе задания. Или вы можете использовать метод `retryUntil`, чтобы определить время, по истечению которого попыток выполнения задания больше не будет.

Если вы не хотите, чтобы задание возвращалось в очередь, если оно ограничено по частоте, вы можете использовать метод `dontRelease`:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new RateLimited('backups'))->dontRelease()];
}
```

Если вы используете Redis, то вы можете использовать посредника `Illuminate\Queue\Middleware\RateLimitedWithRedis`, который лучше настроен для Redis и более

эффективен, чем базовый посредник с ограничением частоты.

## Предотвращение дублирования задания

Laravel включает посредника `Illuminate\Queue\Middleware\WithoutOverlapping`, который позволяет предотвращать перекрытия заданий на основе произвольного ключа. Это может быть полезно, когда задание в очереди изменяет ресурс, который должен изменяться только одним заданием за раз.

Например, представим, что у вас есть задание в очереди, которое обновляет кредитный рейтинг пользователя, и вы хотите предотвратить дублирование задания обновления кредитного рейтинга для одного и того же идентификатора пользователя. Для этого вы можете вернуть посредника `WithoutOverlapping` из метода `middleware` вашего задания:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new WithoutOverlapping($this->user->id)];
}
```

Любые перекрывающиеся задания одного и того же типа будут возвращены в очередь. Можно также указать время в секундах, которое должно пройти до повторной попытки возвращенного задания:

```
/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
```

```
        return [ (new WithoutOverlapping($this->order->id))->releaseAfter(60) ];
    }
```

Если вы хотите немедленно удалить все перекрывающиеся задания, чтобы они не повторялись, вы можете использовать метод `dontRelease`:

```
/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [ (new WithoutOverlapping($this->order->id))->dontRelease() ];
}
```

Посредник `WithoutOverlapping` работает благодаря функции атомарной блокировки Laravel. Но иногда ваше задание может неожиданно завершиться неудачей или таймаутом таким образом, что блокировка не будет освобождена. Поэтому вы можете явно определить время истечения блокировки с помощью метода `expireAfter`. Например, в приведенном ниже примере Laravel даст указание освободить блокировку `WithoutOverlapping` через три минуты после начала обработки задания:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [ (new WithoutOverlapping($this->order->id))->expireAfter(180) ];
}
```

Для посредника `WithoutOverlapping` требуется драйвер кеша, который поддерживает [блокировки](#). В настоящее время драйверы кеша `memcached`, `redis`,

`dynamodb`, `database`, `file`, и `array` поддерживают атомарные блокировки.

## Общий доступ к ключам блокировки

### для разных классов заданий

По умолчанию middleware `WithoutOverlapping` предотвращает перекрытие только заданий одного и того же класса. Таким образом, хотя два разных класса могут использовать один и тот же ключ блокировки, их перекрытие не будет предотвращено. Однако вы можете поручить Laravel применять ключ ко всем классам заданий, используя метод `shared`:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

class ProviderIsDown
{
    // ...

    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}

class ProviderIsUp
{
    // ...

    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}
```

## Ограничение частоты генерации исключений

Laravel содержит посредника `Illuminate\Queue\Middleware\ThrottlesExceptions`, который позволяет вам регулировать вызываемые исключения. Как только задание

вызывает переданное количество исключений, все дальнейшие попытки выполнить задание откладываются до истечения заданного интервала времени. Этот посредник особенно полезен для заданий, которые взаимодействуют с нестабильно работающими сторонними службами.

Например, представим себе задание в очереди, взаимодействующее со сторонним API, который начинает выбрасывать исключения. Чтобы ограничить исключения, вы можете вернуть посредника `ThrottlesExceptions` из метода `middleware` вашего задания. Как правило, этот посредник должен быть связан с заданием, которое реализует [попытки, основанные на времени](#):

```
use DateTime;
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new ThrottlesExceptions(10, 5 * 60)];
}

/**
 * Задать временной предел попыток выполнить задания.
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(30);
}
```

Первый аргумент конструктора посредника — это количество исключений, которые задание может выбросить перед ограничением. Второй аргумент конструктора — это количество секунд, которое должно пройти, прежде чем будет предпринято повторное выполнение задания после его ограничения. В приведенном выше примере кода, если задание выбросит 10 последовательных исключений, мы подождем 5 минут перед его повторной попыткой выполнения, ограниченную 30-минутным лимитом времени.

Когда задание вызывает исключение, но порог исключения еще не достигнут, то задание обычно немедленно повторяется. Однако вы можете указать количество

минут, на которые такое задание должно быть отложено, вызвав метод `backoff` при определении метода `middleware`:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 5 * 60))->backoff(5)];
}
```

Внутренне этот посредник использует систему кеширования Laravel для реализации ограничений частоты, а имя класса задания используется в качестве «ключа» кеша. Вы можете переопределить этот ключ, вызвав метод `by` при определении метода `middleware` вашего задания. Это может быть полезно, если у вас есть несколько заданий, взаимодействующих с одной и той же сторонней службой, и вы хотите, чтобы у них была общая «корзина» ограничений:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Получить посредника, через которого должно пройти задание.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 10 * 60))->by('key')];
}
```

По умолчанию это промежуточное программное обеспечение будет регулировать каждое исключение. Вы можете изменить это поведение, вызвав метод `when` при подключении посредника к вашему заданию. Исключение будет регулироваться только в том случае, если закрытие, предоставленное методу `when`, вернет `true`:

```
use Illuminate\Http\Client\HttpClientException;
use Illuminate\Queue\Middleware\ThrottlesExceptions;
```

```
/**  
 * Get the middleware the job should pass through.  
 *  
 * @return array<int, object>  
 */  
public function middleware(): array  
{  
    return [(new ThrottlesExceptions(10, 10 * 60))->when(  
        fn (Throwable $throwable) => $throwable instanceof HttpClientException  
    )];  
}
```

Если вы хотите, чтобы регулируемые исключения сообщались обработчику исключений вашего приложения, вы можете сделать это, вызвав метод `report` при подключении посредника к вашему заданию. При желании вы можете предоставить замыкание для метода `report`, и об исключении будет сообщено только в том случае, если данное замыкание возвращает `true`:

```
use Illuminate\Http\Client\HttpClientException;  
use Illuminate\Queue\Middleware\ThrottlesExceptions;  
  
/**  
 * Get the middleware the job should pass through.  
 *  
 * @return array<int, object>  
 */  
public function middleware(): array  
{  
    return [(new ThrottlesExceptions(10, 10 * 60))->report(  
        fn (Throwable $throwable) => $throwable instanceof HttpClientException  
    )];  
}
```

Если вы используете Redis в качестве драйвера кеша вашего приложения, то вы можете использовать класс [Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis](#). Этот класс более эффективен при управлении ограничениями исключений с помощью Redis.

## Пропуск заданий

Посредник `Skip` позволяет вам указать, что задание должно быть пропущено/удалено без необходимости изменения логики задания. Метод `Skip::when` удаляет задание, если данное условие оценивается как `true`, а метод `Skip::unless` удаляет задание, если условие оценивается как `false`:

```
use Illuminate\Queue\Middleware\Skip;

/**
 * Get the middleware the job should pass through.
 */
public function middleware(): array
{
    return [
        Skip::when($someCondition),
    ];
}
```

Вы также можете передать `Closure` методам `when` и `unless` для более сложной условной оценки:

```
use Illuminate\Queue\Middleware\Skip;

/**
 * Get the middleware the job should pass through.
 */
public function middleware(): array
{
    return [
        Skip::when(function (): bool {
            return $this->shouldSkip();
        }),
    ];
}
```

## # Отправка заданий

После того как вы написали свой класс задания, вы можете отправить его, используя метод `dispatch` самого задания. Аргументы, переданные методу `dispatch`, будут переданы конструктору задания:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Сохранить новый подкаст.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast);

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

Если требуется отправить задание по условию, то можно использовать методы `dispatchIf` и `dispatchUnless`:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);

ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

В новых приложениях Laravel драйвер `sync` является драйвером очереди по-умолчанию. Этот драйвер выполняет задания синхронно во время запроса, что часто бывает удобно при локальной разработке. Если вы действительно хотите поставить задания в очередь для фоновой обработки, вы можете указать другой драйвер очереди в файле конфигурации вашего приложения `config/queue.php`.

## Отложенная отправка

Если вы хотите указать, что задание не должно быть немедленно доступно для обработчика очереди, вы можете использовать метод `delay` при отправке задания. Например, давайте укажем, что задание не должно быть доступно для обработки в течение 10 минут после его отправки:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Сохранить новый подкаст.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast)
            ->delay(now()->addMinutes(10));

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

В некоторых случаях для заданий может быть настроена задержка по умолчанию. Если вам нужно обойти эту задержку и отправить задание на немедленную обработку, вы можете использовать метод `withoutDelay`:

```
ProcessPodcast::dispatch($podcast)->withoutDelay();
```

У сервиса очередей Amazon SQS максимальное

время задержки составляет 15 минут.

## Отправка задания после отправки ответа в браузер

В качестве альтернативы, метод `dispatchAfterResponse` задерживает отправку задания до тех пор, пока HTTP-ответ не будет отправлен в браузер пользователя, если ваш веб-сервер использует FastCGI. Это по прежнему позволит пользователю получить ответ от приложения, даже если задание в очереди все еще выполняется. Обычно это следует использовать только для заданий, которые занимают около секунды, например, для отправки электронного письма. Поскольку они обрабатываются в рамках текущего HTTP-запроса, отправляемые таким образом задания не требуют запуска обработчика очереди для их обработки:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

Вы также можете отправить замыкание и связать метод `afterResponse` с помощником `dispatch`, чтобы выполнить функцию после того, как HTTP-ответ был отправлен в браузер:

```
use App\Mail\WelcomeMessage;
use Illuminate\Support\Facades\Mail;

dispatch(function () {
    Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

## Синхронная отправка

Если вы хотите отправить задание немедленно (синхронно), то вы можете использовать метод `dispatchSync`. При использовании этого метода задание не будет поставлено в очередь и будет выполнено немедленно в рамках текущего процесса:

```
<?php

namespace App\Http\Controllers;
```

```

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Сохранить новый подкаст.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Создание подкаста ...

        ProcessPodcast::dispatchSync($podcast);

        return redirect('/podcasts');
    }
}

```

## Задания и транзакции базы данных

Хотя отправлять задания в рамках транзакций базы данных вполне нормально, вам следует уделить особое внимание тому, чтобы ваше задание действительно могло выполняться успешно. При отправке задания в то время, как открыта транзакция в базе данных, возможно, что задание будет обработано до фиксации родительской транзакции. Когда это происходит, любые обновления, внесенные вами в модели или записи базы данных во время транзакции базы данных, могут еще не быть отражены в базе данных. Кроме того, любые модели или записи базы данных, созданные в рамках транзакции, могут даже не существовать в базе данных.

К счастью, Laravel содержит несколько методов решения этой проблемы. Во-первых, вы можете задать параметр соединения `after_commit` в массиве конфигурации соединения к очереди:

```

'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],

```

Когда параметр `after_commit` имеет значение `true`, вы можете отправлять задания в транзакциях базы данных; однако, Laravel будет ждать, пока все открытые родительские транзакции базы данных будут завершены, прежде чем фактически отправить задание. Если в настоящее время нет открытых транзакций, задание будет отправлено немедленно.

При откате транзакции из-за исключения, возникшего во время транзакции, отправленные во время этой транзакции задания будут отброшены.

Установка параметру конфигурации `after_commit` значения `true` также вызовет отправку всех поставленных в очередь слушателей событий, почтовых отправлений, уведомлений и широковещательных событий после того, как все открытые транзакции базы данных были зафиксированы.

## Непосредственное указание поведения отправки при фиксации транзакций БД

Если вы не установите для параметра конфигурации соединения очереди `after_commit` значение `true`, то вы все равно можете указать, что конкретное задание должно быть отправлено после того, как все открытые транзакции базы данных будут завершены. Для этого вы можете связать метод `afterCommit` с операцией отправки:

```
use App\Jobs\ProcessPodcast;  
  
ProcessPodcast::dispatch($podcast)->afterCommit();
```

Аналогично, если для параметра конфигурации `after_commit` установлено значение `true`, вы можете указать, что конкретное задание должно быть отправлено немедленно, не дожидаясь завершения каких-либо открытых транзакций базы данных:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

## Цепочка заданий

Цепочка заданий позволяет указать список заданий в очереди, которые должны выполняться последовательно после успешного выполнения основного задания. Если одно задание в последовательности завершается неуспешно, то остальные задания не выполняются. Чтобы выполнить цепочку заданий в очереди, вы можете использовать метод `chain`, фасада `Bus`. Командная шина Laravel – это компонент нижнего уровня, на котором построена диспетчеризация заданий в очереди:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->dispatch();
```

В дополнение к цепочке экземпляров класса задания вы также можете передавать функции:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    function () {
        Podcast::update(/* ... */);
    },
])->dispatch();
```

Удаление заданий с помощью метода `$this->delete()` внутри задания не остановит обработку связанных заданий. Цепочка прекратит выполнение только в случае сбоя задания в цепочке.

## Соединения и очередь цепочки заданий

Если вы хотите указать соединение и очередь, которые должны использоваться для связанных заданий, вы можете использовать методы `onConnection` и `onQueue`. Эти методы указывают соединение и имя очереди, которые следует использовать, если заданию явно не назначено другое соединение / очередь:

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

## Добавление заданий в цепочку

Иногда вам может потребоваться добавить задание в существующую цепочку заданий из другого задания в этой цепочке. Вы можете сделать это, используя методы `prependToChain` и `appendToChain`:

```
/**
 * Execute the job.
 */
public function handle(): void
{
    // ...

    // Prepend to the current chain, run job immediately after current job...
    $this->prependToChain(new TranscribePodcast);

    // Append to the current chain, run job at end of chain...
    $this->appendToChain(new TranscribePodcast);
}
```

## Отказы в цепочке заданий

При объединении заданий в цепочку вы можете использовать метод `catch`, чтобы указать функцию, которая должна вызываться, если задание в цепочке завершается неуспешно. Данная функция получит экземпляр `Throwable`, спровоцировавшего провал задания:

```
use Illuminate\Support\Facades\Bus;
use Throwable;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->catch(function (Throwable $e) {
    // Задание в цепочке не выполнено ...
})->dispatch();
```

Поскольку функции-замыкания сериализуются и выполняются позже в очереди Laravel, вам не следует использовать `$this` в замыканиях.

## Настройка соединения и очереди

### Отправка в определенную очередь

Помещая задания в разные очереди, вы можете «классифицировать» свои задания в очереди и даже определять приоритеты, сколько обработчиков вы назначаете в разные очереди. Имейте в виду, что при этом задания не отправляются в разные «соединения» очередей, как определено в файле конфигурации очереди, а только в определенные очереди в рамках одного соединения. Чтобы указать очередь, используйте метод `onQueue` при отправке задания:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Сохранить новый подкаст.
}
```

```

    */
public function store(Request $request): RedirectResponse
{
    $podcast = Podcast::create(/* ... */);

    // Создание подкаста ...

    ProcessPodcast::dispatch($podcast)->onQueue('processing');

    return redirect('/podcasts');
}
}

```

Кроме того, вы можете указать очередь задания, вызвав метод `onQueue` в конструкторе задания:

```

<?php

namespace App\Jobs;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;

class ProcessPodcast implements ShouldQueue
{
    use Queueable;

    /**
     * Создать новый экземпляр задания.
     */
    public function __construct()
    {
        $this->onQueue('processing');
    }
}

```

## Отправка в конкретное соединение

Если ваше приложение взаимодействует с несколькими соединениями очередей, то вы можете указать, на какое соединение отправить задание, используя метод `onConnection`:

```
<?php
```

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Сохранить новый подкаст.
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Создание подкаста ...

        ProcessPodcast::dispatch($podcast)->onConnection('sqS');

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

Вы можете связать методы `onConnection` и `onQueue` вместе, чтобы указать соединение и очередь для задания:

```
ProcessPodcast::dispatch($podcast)
    ->onConnection('sqS')
    ->onQueue('processing');
```

Кроме того, вы можете указать соединение задания, вызвав метод `onConnection` в конструкторе задания:

```
<?php

namespace App\Jobs;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;

class ProcessPodcast implements ShouldQueue
{
    use Queueable;
```

```
/**
 * Создать новый экземпляр задания.
 */
public function __construct()
{
    $this->onConnection('sqS');
}
}
```

## Указание максимального количества попыток задания / значений тайм-аута

### Максимальное количество попыток

Если в одном из ваших заданий в очереди обнаруживается ошибка, то вы, вероятно, не хотите, чтобы оно продолжало повторять попытки бесконечно. Laravel предлагает различные способы указать, сколько раз и как долго задание может быть повторно выполняться.

Один из подходов к указанию максимального количества попыток выполнения задания – это использование переключателя `--tries` в командной строке Artisan. Это будет применяться ко всем заданиям обработчика, если только в обрабатываемом задание не указано количество попыток его выполнения:

```
php artisan queue:work --tries=3
```

Если задание превышает максимальное количество попыток, то оно будет считаться «неудачным». Для получения дополнительной информации об обработке невыполненных заданий обратитесь к [документации по разбору неудачных заданий](#). Если указано `--tries=0` в команде `queue:work`, задание будет повторяться бесконечно.

Вы можете применить более детальный подход, указав максимальное количество попыток выполнения задания для самого класса задания. Если для задания указано максимальное количество попыток, оно будет иметь приоритет над значением `--tries`, указанным в командной строке:

```
<?php
```

```
namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * Количество попыток выполнения задания.
     *
     * @var int
     */
    public $tries = 5;
}
```

Если вам необходимо динамически управлять максимальным числом попыток выполнения конкретного задания, вы можете определить метод `tries` внутри задания:

```
/**
 * Определите количество попыток выполнения задания.
 */
public function tries(): int
{
    return 5;
}
```

## Попытки, основанные на времени

В качестве альтернативы определению количества попыток выполнения задания до того, как оно завершится ошибкой, вы можете определить время, когда прекратить попытки выполнения задания. Это позволяет выполнять задание любое количество раз в течение заданного периода времени. Чтобы определить время, через которое больше не следует пытаться выполнить задание, добавьте метод `retryUntil` в свой класс задания. Этот метод должен возвращать экземпляр `DateTime`:

```
use DateTime;

/**
 * Задать временной предел попыток выполнить задания.
 *
 * @return \DateTime
 */
public function retryUntil(): DateTime
```

```
{  
    return now()->addMinutes(10);  
}
```

Вы также можете определить свойство `$tries` или метод `retryUntil` в ваших [слушателях событий](#).

## Максимальное количество исключений

Иногда вы можете указать, что задание может быть выполнено много раз, но должно завершиться ошибкой, если повторные попытки инициированы заданным количеством необработанных исключений (в отличие от отправки напрямую методом `release`). Для этого вы можете определить свойство `maxExceptions` в своем классе задания:

```
<?php  
  
namespace App\Jobs;  
  
use Illuminate\Support\Facades\Redis;  
  
class ProcessPodcast implements ShouldQueue  
{  
    /**  
     * Количество попыток выполнения задания.  
     *  
     * @var int  
     */  
    public $tries = 25;  
  
    /**  
     * Максимальное количество разрешенных необработанных исключений.  
     *  
     * @var int  
     */  
    public $maxExceptions = 3;  
  
    /**  
     * Выполнить задание.  
     */  
    public function handle(): void  
    {
```

```
Redis::throttle('key')->allow(10)->every(60)->then(function () {
    // Блокировка получена, обрабатываем подкаст ...
}, function () {
    // Невозможно получить блокировку ...
    return $this->release(10);
});
}
}
```

В этом примере задание высвобождается на десять секунд, если приложение не может получить блокировку Redis, и будет продолжать повторяться до 25 раз. Однако задание завершится ошибкой, если оно вызовет три необработанных исключения.

## Таймаут

Часто вы приблизительно знаете, сколько времени займет выполнение заданий в очереди. По этой причине Laravel позволяет вам указать значение «таймаута». По умолчанию значение таймаута составляет 60 секунд. Если задание обрабатывается дольше, чем количество секунд, указанное в значении тайм-аута, рабочий процесс, обрабатывающий задание, завершит работу с ошибкой. Обычно worker перезапускается автоматически [менеджером процессов, настроенным на вашем сервере](#).

Максимальное количество секунд, в течение которых могут выполняться задания, можно указать с помощью переключателя `--timeout` в командной строке Artisan:

```
php artisan queue:work --timeout=30
```

Если задание превышает максимальное количество попыток из-за постоянного тайм-аута, оно будет помечено как «неудачное».

Вы также можете определить таймаут в самом классе задания. В этом случае это значение будет иметь приоритет над любым таймаутом, указанным в командной строке:

```
<?php
namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
```

```
{  
    /**  
     * Количество секунд, в течение которых задание может выполняться до истечения таймаута.  
     *  
     * @var int  
     */  
    public $timeout = 120;  
}
```

Иногда процессы блокировки ввода-вывода, такие, как сокеты или исходящие HTTP-соединения, могут не учитывать указанный вами таймаут. Следовательно, при использовании этих функций вы всегда должны пытаться указать таймаут, используя их API. Например, при использовании Guzzle вы всегда должны [запроса](#) указывать значение таймаута соединения и запроса.

Для указания тайм-аутов заданий необходимо установить PHP-расширение `pcntl`. Кроме того, значение тайм-аута в задании всегда должно быть меньше значения `"retry after"`. В противном случае задание может быть повторено до того, как оно фактически завершится или истечет время ожидания.

## Неудача заданий по таймауту

Если вы хотите указать, что задание должно быть помечено как `failed` по истечении времени, вы можете определить свойство `$failOnTimeout` для класса задания:

```
/**  
 * Indicate if the job should be marked as failed on timeout.  
 *  
 * @var bool  
 */  
public $failOnTimeout = true;
```

## Обработка ошибок

Если во время обработки задания возникает исключение, задание автоматически возвращается в очередь (release), чтобы его можно было повторить. Задание будет продолжать возвращаться до тех пор, пока оно не будет выполнено максимальное количество раз, разрешенное вашим приложением. Максимальное количество попыток определяется переключателем `--tries`, используемым в команде `queue:work` Artisan. В качестве альтернативы максимальное количество попыток может быть определено в самом классе задания. Более подробную информацию о запуске обработчика очереди [можно найти ниже](#).

## Ручное освобождение задания

По желанию можно вручную вернуть задание в очередь, чтобы его можно было повторить позже. Вы можете сделать это, вызвав метод `release`:

```
/**
 * Выполнить задание.
 */
public function handle(): void
{
    // ...

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

По умолчанию метод `release` помещает задание обратно в очередь для немедленной обработки. Однако вы можете указать очереди не делать задание доступным для обработки до тех пор, пока не истечет заданное количество секунд, передав целое число или экземпляр даты в методе `release`:

```
$this->release(10);

$this->release(now()->addSeconds(10));
```

## Пометка задания неудачным

Иногда требуется вручную пометить задание как «неудачное». Для этого вы можете вызвать метод `fail`:

```
/**
 * Выполнить задание.
```

```
 */
public function handle(): void
{
    // ...

    $this->fail();
}
```

Если вы хотите пометить свою работу как неудавшуюся из-за обнаруженного исключения, то вы можете передать исключение методу `fail`. Или, для удобства, вы можете передать строковое сообщение об ошибке, которое будет преобразовано для вас в исключение:

```
$this->fail($exception);

$this->fail('Something went wrong.');
```

Для получения дополнительной информации об обработке невыполненных заданий обратитесь к [документации по разбору неудачных заданий](#).

## # Пакетная обработка заданий

Функционал пакетной обработки заданий Laravel позволяет вам легко выполнить пакет заданий, по завершению которого дополнительно совершить определенные действия. Перед тем, как начать, вы должны создать миграцию базы данных, чтобы построить таблицу, содержащую метаинформацию о ваших пакетах заданий, такую как процент их завершения. Эта миграция может быть сгенерирована с помощью команды `make:queue-batches-table` Artisan:

```
php artisan make:queue-batches-table

php artisan migrate
```

## Определение пакета заданий

Чтобы определить задание с возможностью пакетной передачи, вы, как обычно, должны [создать задание в очереди](#); тем не менее, вы должны добавить к классу задания трейт `Illuminate\Bus\Batchable`. Этот трейт обеспечивает доступ к методу `batch`, использующийся для получения текущего пакета, в котором выполняется задание:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;

class ImportCsv implements ShouldQueue
{
    use Batchable, Queueable;

    /**
     * Выполнить задание.
     *
     * @return void
     */
    public function handle(): void
    {
        if ($this->batch()->cancelled()) {
            // Определяем, был ли пакет отменен ...

            return;
        }

        // Импортируем часть CSV-файла ...
    }
}
```

## Отправка пакета заданий

Чтобы отправить пакет заданий, вы должны использовать метод `batch` фасада `Bus`. Основное преимущество обработки заданий одним пакетом – в том, что можно выполнить некий код по завершению этого пакета. Этот код добавляется в виде функций в аргументах методов `then`, `catch` и `finally`. Каждая из этих функций получит при вызове экземпляр `Illuminate\Bus\Batch`. В этом примере мы представим, что отправляем в очередь пакет заданий, каждое из которых обрабатывает указанное количество строк из файла CSV:

```
use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->before(function (Batch $batch) {
    // Пакет заданий создан, но не добавлено ни одно задание ...
})->progress(function (Batch $batch) {
    // Одна задача успешно завершена ...
})->then(function (Batch $batch) {
    // Все задания успешно завершены ...
})->catch(function (Batch $batch, Throwable $e) {
    // Обнаружено первое проваленное задание из пакета ...
})->finally(function (Batch $batch) {
    // Завершено выполнение пакета ...
})->dispatch();

return $batch->id;
```

Идентификатор пакета, к которому можно получить доступ через свойство `$batch->id`, можно использовать для [запроса к команднойшине Laravel](#) для получения информации о пакете после того, как он был отправлен.

Поскольку пакетные обратные вызовы сериализуются и выполняются позднее в очереди Laravel, вы не должны использовать переменную `$this` в обратных вызовах. Кроме того, поскольку пакетные задания заключены в транзакции базы данных, операторы базы данных, вызывающие неявную фиксацию, не должны выполняться внутри заданий.

## Именованные пакеты заданий

Некоторые инструменты, такие как Laravel Horizon и Laravel Telescope, могут предоставлять более удобную для пользователя отладочную информацию о пакетах, если пакеты имеют имена. Чтобы присвоить пакету произвольное имя, вы можете вызвать метод `name` при определении пакета:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // Все задания успешно завершены ...
})->name('Import CSV')->dispatch();
```

## Соединение и очередь пакета

Если вы хотите указать соединение и очередь, которые должны использоваться для пакетных заданий, то вы можете использовать методы `onConnection` и `onQueue`. Все пакетные задания должны выполняться в одном соединении и в одной очереди:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // Все задания успешно завершены ...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

## Цепочки заданий (Chains) и Пакеты (Batches)

Вы можете определить набор [связанных заданий](#) в пакете, поместив связанные задания в массив. Например, мы можем выполнить две цепочки заданий параллельно и выполнить замыкание, когда обе цепочки заданий завершат обработку:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [

```

```
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

И наоборот, вы можете запускать пакеты заданий внутри [цепочки](#), определяя пакеты внутри цепочки. Например, вы можете сначала запустить пакет заданий для выпуска нескольких подкастов, а затем пакет заданий для отправки уведомлений о выпуске:

```
use App\Jobs\FlushPodcastCache;
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new FlushPodcastCache,
    Bus::batch([
        new ReleasePodcast(1),
        new ReleasePodcast(2),
    ]),
    Bus::batch([
        new SendPodcastReleaseNotification(1),
        new SendPodcastReleaseNotification(2),
    ]),
])->dispatch();
```

## Добавление заданий в пакет заданий

Иногда может быть полезно добавить дополнительные задания в пакет, непосредственно из задания, уже находящегося в пакете. Этот шаблон может быть полезен, когда вам нужно выполнить пакетную обработку тысяч заданий, выполнение которых может занять слишком много времени во время веб-запроса, когда формируется пакет. Таким образом, вместо этого вы можете отправить начальный пакет заданий «загрузчику», которые дополнят пакет еще большим количеством заданий:

```
$batch = Bus::batch([
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
```

```
])->then(function (Batch $batch) {
    // Все задания успешно завершены ...
})->name('Import Contacts')->dispatch();
```

В этом примере мы будем использовать задание `LoadImportBatch`, чтобы дополнить пакет дополнительными заданиями. Для этого мы можем использовать метод `add` экземпляра пакета, к которому можно получить доступ через метод `batch` задания:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * Выполнить задание.
 */
public function handle(): void
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```

Вы можете добавлять задания в пакет только из задания, которое принадлежит к тому же пакету.

## Инспектирование пакета

Экземпляр `Illuminate\Bus\Batch`, который передается замыканиям по завершению пакета, имеет множество свойств и методов, помогающих взаимодействовать с данным пакетом заданий и его анализа:

```
// UUID пакета ...
$batch->id;

// Название пакета (если применимо) ...
$batch->name;
```

```
// Количество заданий, назначенных пакету ...
$batch->totalJobs;

// Количество заданий, которые не были обработаны очередью ...
$batch->pendingJobs;

// Количество неудачных заданий ...
$batch->failedJobs;

// Количество заданий, обработанных на данный момент ...
$batch->processedJobs();

// Процент завершения пакетной обработки (0-100) ...
$batch->progress();

// Указывает, завершено ли выполнение пакета ...
$batch->finished();

// Отменить выполнение пакета ...
$batch->cancel();

// Указывает, был ли пакет отменен ...
$batch->cancelled();
```

## Возврат пакетов заданий из маршрутов

Все экземпляры `Illuminate\Bus\Batch` являются сериализуемыми в формате JSON, что означает, что вы можете возвращать их непосредственно из одного из маршрутов вашего приложения, чтобы получить JSON, содержащий информацию о пакете, включая ход его завершения. Это позволяет удобно отображать информацию о ходе выполнения пакета в пользовательском интерфейсе вашего приложения.

Чтобы получить пакет по его идентификатору, вы можете использовать метод `findBatch` фасада `Bus`:

```
use Illuminate\Support\Facades\Bus;
use Illuminate\Support\Facades\Route;

Route::get('/batch/{batchId}', function (string $batchId) {
    return Bus::findBatch($batchId);
});
```

## Отмена пакетов

Иногда требуется отменить выполнение определенного пакета. Это можно сделать, вызвав метод `cancel` экземпляра `Illuminate\Bus\Batch`:

```
/**
 * Выполнить задание.
 */
public function handle(): void
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

    if ($this->batch()->cancelled()) {
        return;
    }
}
```

Как вы, возможно, заметили в предыдущих примерах, пакетные задания обычно должны определить, был ли соответствующий пакет отменен, прежде чем продолжить выполнение. Однако для удобства вместо этого вы можете назначить заданию `SkipIfBatchCancelled middleware`. Как следует из названия, это middleware будет инструктировать Laravel не обрабатывать задание, если соответствующий пакет был отменен:

```
use Illuminate\Queue\Middleware\SkipIfBatchCancelled;

/**
 * Get the middleware the job should pass through.
 */
public function handle(): array
{
    return [new SkipIfBatchCancelled];
}
```

## Отказы в пакете заданий

Если задание в пакете завершается неуспешно, то будет вызвано замыкание `catch` (если назначено). Это замыкание вызывается только для первого проваленного задания в пакете.

## Допущение отказов

Когда задание в пакете завершается неуспешно, Laravel автоматически помечает пакет как «отмененный». При желании вы можете отключить это поведение, чтобы при провале задания пакет не отмечался автоматически как отмененный. Это может быть выполнено путем вызова метода `allowFailures` при отправке пакета:

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // Все задания успешно завершены ...
})->allowFailures()->dispatch();
```

## Повторная попытка выполнения

### неудачных пакетных заданий

Для удобства Artisan содержит команду `queue:retry-batch`, которая позволяет вам легко повторить все неудачные задания для указанного пакета.

Команда `queue:retry-batch` принимает UUID пакета, чьи неудачные задания следует повторить:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

## Очистка пакетов

Если не применять очистку, то таблица `job_batches` может очень быстро накапливать записи. Чтобы избежать этого, вы должны запланировать ежедневный запуск команды `queue:prune-batches` Artisan:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('queue:prune-batches')->daily();
```

По умолчанию все готовые пакеты, возраст которых превышает 24 часа, будут удалены. Вы можете использовать параметр `hours` при вызове команды, чтобы определить, как долго хранить пакетные данные. Например, следующая команда удалит все пакеты, завершенные более 48 часов назад:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('queue:prune-batches --hours=48')->daily();
```

Иногда в таблице `jobs_batches` могут накапливаться записи пакетов, которые так и не были успешно завершены, например, пакеты, в которых задание не удалось выполнить, и это задание так и не было успешно перезапущено. Вы можете поручить команде `queue:prune-batches` очистить эти незавершенные пакетные записи, используя опцию `unfinished`:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('queue:prune-batches --hours=48 --unfinished=72')->daily();
```

Аналогично, ваша таблица `jobs_batches` может также накапливать записи об отмененных пакетах. Вы можете указать команде `queue:prune-batches` удалить эти отмененные пакетные записи, используя флаг `cancelled`:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('queue:prune-batches --hours=48 --cancelled=72')->daily();
```

## Хранение пакетов в DynamoDB

Laravel также поддерживает хранение мета-информации о пакетах в [DynamoDB](#), а не в реляционной базе данных. Однако вам придется вручную создать таблицу DynamoDB для хранения всех записей о пакетах.

Обычно эта таблица должна называться `job_batches`, но вы можете назвать таблицу в зависимости от значения конфигурации `queue.batching.table` в файле конфигурации очереди вашего приложения.

## Конфигурация таблицы пакетов DynamoDB

Таблица `job_batches` должна иметь строковый первичный ключ с именем `application` и строковый первичный ключ с именем `id`. Часть `application` ключа будет содержать имя вашего приложения, как определено значением `name` в файле конфигурации приложения `app`. Поскольку имя приложения является частью ключа

таблицы DynamoDB, вы можете использовать ту же таблицу для хранения пакетов задач для нескольких приложений Laravel.

Кроме того, вы можете определить атрибут `ttl` для вашей таблицы, если хотите воспользоваться [автоматической обрезкой пакетов](#).

## Конфигурация DynamoDB

Затем установите AWS SDK, чтобы ваше Laravel-приложение могло взаимодействовать с Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Затем установите значение параметра конфигурации `queue.batching.driver` на `dynamodb`. Кроме того, вам следует определить параметры конфигурации `key`, `secret` и `region` в массиве конфигурации `batching`. Эти параметры будут использоваться для аутентификации в AWS. При использовании драйвера `dynamodb` параметр конфигурации `queue.batching.database` не требуется:

```
'batching' => [
    'driver' => env('QUEUE_BATCHING_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'job_batches',
],
```

## Очистка пакетов в DynamoDB

При использовании [DynamoDB](#) для хранения информации о пакетах задач, типичные команды очистки для пакетов не будут работать. Вместо этого вы можете использовать [встроенную функцию TTL в DynamoDB](#) для автоматического удаления записей о старых пакетах.

Если вы определили таблицу DynamoDB с атрибутом `ttl`, вы можете определить параметры конфигурации, чтобы указать Laravel, как удалять записи о пакетах. Значение параметра конфигурации `queue.batching.ttl_attribute` определяет имя атрибута, содержащего TTL, а значение параметра конфигурации `queue.batching.ttl`

определяет количество секунд, через которое запись о пакете может быть удалена из таблицы DynamoDB, относительно последнего времени обновления записи:

```
'batching' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'job_batches',
    'ttl_attribute' => 'ttl',
    'ttl' => 60 * 60 * 24 * 7, // 7 days...
],
]
```

## # Анонимные очереди

Вместо отправки класса задания в очередь вы также можете отправить функцию. Это отлично подходит для быстрых и простых задач, которые необходимо выполнять вне текущего цикла запроса. При отправке функции в очередь содержимое кода функции криптографически подписывается, поэтому его нельзя изменить при передаче:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

Используя метод `catch`, вы можете определить функцию, которая должна быть выполнена, если анонимная очередь не завершится успешно после исчерпания всех сконфигурированных попыток повтора вашей очереди:

```
use Throwable;

dispatch(function () use ($podcast) {
    $podcast->publish();
})->catch(function (Throwable $e) {
    // Это задание завершилось неудачно ...
});
```

Поскольку функции-замыкания в `catch` сериализуются и выполняются очередью Laravel позднее, вам не следует использовать `$this` в обратных вызовах `catch`.

## # Запуск обработчика очереди

### Команда `queue:work`

Laravel включает команду Artisan, которая запускает обработчика очереди и обрабатывает новые задания по мере их помещения в очередь. Вы можете запустить обработчик с помощью команды `queue:work` Artisan. Обратите внимание, что после запуска команды `queue:work` она будет продолжать работать, пока не будет остановлена вручную или пока вы не закроете терминал (консоль):

```
php artisan queue:work
```

Чтобы процесс `queue:work` работал постоянно в фоновом режиме, вам следует использовать диспетчер процессов, такой как [Supervisor](#), чтобы гарантировать, что worker очереди не перестанет работать.

Вы можете включить флаг `-v` при вызове команды `queue:work`, если хотите, чтобы идентификаторы обработанных заданий были включены в output команды:

```
php artisan queue:work -v
```

Помните, что обработчики очереди – это долгоживущие процессы, которые хранят состояние загруженного приложения в памяти. В результате они не заметят изменений в вашей кодовой базе после их запуска. Итак, во время процесса развертывания обязательно [перезапустите своих обработчиков очереди](#). Кроме

того, помните, что любое статическое состояние, созданное или измененное вашим приложением, не будет автоматически пробрасываться между заданиями.

Как вариант, вы можете запустить команду `queue:listen`. При использовании команды `queue:listen` вам не нужно вручную перезапускать обработчик, если вы хотите перезагрузить обновленный код или сбросить состояние приложения; однако эта команда значительно менее эффективна, чем команда `queue:work`:

```
php artisan queue:listen
```

## Запуск нескольких обработчиков очереди

Чтобы назначить несколько обработчиков в очередь и обрабатывать задания одновременно, вы должны просто запустить несколько процессов `queue:work`. Это можно сделать либо локально с помощью нескольких вкладок в вашем терминале, либо в эксплуатационном режиме, используя параметры конфигурации вашего диспетчера процессов. [При использовании Supervisor](#) вы можете использовать значение конфигурации `numprocs`.

## Указание соединения и очереди

Вы также можете указать, какое соединение очереди должен использовать обработчик. Имя соединения, переданное команде `work`, должно соответствовать одному из соединений, определенных в конфигурационном файле `config/queue.php`:

```
php artisan queue:work redis
```

По умолчанию команда `queue:work` обрабатывает задания только для очереди по умолчанию на данном соединении. Однако, вы можете дополнительно указать, какие очереди необходимо обрабатывать для указанного соединения. Например, если все ваши электронные письма обрабатываются в очереди `emails` соединения `redis`, то вы можете использовать команду, чтобы запустить обработчик только для этой очереди:

```
php artisan queue:work redis --queue=emails
```

## Обработка указанного количества заданий

Переключатель `--once` обработчика используется для указания обработать только одно задание из очереди:

```
php artisan queue:work --once
```

Параметр `--max-jobs` обработчика проинструктирует его обработать заданное количество заданий, а затем выйти. Этот параметр может быть полезен в сочетании с [Supervisor](#), чтобы ваши рабочие процессы автоматически перезапускались после обработки заданного количества заданий, освобождая любую занятую ими память:

```
php artisan queue:work --max-jobs=1000
```

## Обработка всех заданий в очереди с последующим выходом

Переключатель `--stop-when-empty` обработчика может использоваться, чтобы дать ему указание обработать все задания и затем корректно завершить работу. Этот параметр может быть полезен при обработке очередей Laravel в контейнере Docker, если вы хотите выключить контейнер после того, как очередь пуста:

```
php artisan queue:work --stop-when-empty
```

## Обработка заданий за заданное количество секунд

Параметр `--max-time` обработчика может использоваться, чтобы дать ему указание обрабатывать задания в течение заданного количества секунд, а затем выйти. Этот параметр может быть полезен в сочетании с [Supervisor](#), чтобы ваши рабочие процессы автоматически перезапускались после обработки заданий в течение заданного времени, освобождая любую занятую ими память:

```
# Process jobs for one hour and then exit...
php artisan queue:work --max-time=3600
```

## Продолжительность задержки выполнения обработчика

Когда задания доступны в очереди, обработчик будет продолжать обрабатывать задания без задержки между ними. Однако опция `sleep` определяет, сколько секунд обработчик будет «спать», если нет новых доступных заданий. Конечно, во время задержки выполнения обработчик не будет обрабатывать никаких новых заданий – задания будут обработаны после того, как обработчик снова проснется:

```
php artisan queue:work --sleep=3
```

## Режим обслуживания и очереди

Пока ваше приложение находится в [режиме обслуживания](#), задания, поставленные в очередь, не будут обрабатываться. После выхода приложения из режима обслуживания задания будут обрабатываться в обычном режиме.

Чтобы обрабатывать задания в очереди, даже если включён режим обслуживания, вы можете использовать опцию `--force`:

```
php artisan queue:work --force
```

## Соображения относительно ресурсов

Демоны обработчиков очередей не «перезагружают» фреймворк перед обработкой каждого задания. Следовательно, вы должны освобождать все тяжелые ресурсы после завершения каждого задания. Например, если вы выполняете манипуляции с изображениями с помощью библиотеки GD, вы должны освободить память с помощью `imagedestroy`, когда вы закончите обработку изображения.

## Приоритеты очереди

Иногда вы можете установить приоритетность обработки очередей. Например, в конфигурационном файле `config/queue.php` для очереди по умолчанию вашего соединения `redis` вы можете установить `low`. По желанию можно поместить задание в очередь с «высоким» (`high`) приоритетом, например:

```
dispatch((new Job)->onQueue('high'));
```

Чтобы запустить обработчика, который проверяет, что все задания очереди `high` обработаны, прежде чем переходить к любым заданиям в очереди `low`, передайте разделенный запятыми список имен очередей команде `work`:

```
php artisan queue:work --queue=high,low
```

## Обработчики очереди и развертывание

Поскольку обработчики очереди – это долгоживущие процессы, они не заметят изменений в вашем коде без перезапуска. Итак, самый простой способ развернуть приложение с использованием обработчиков очереди – это перезапустить обработчиков во время процесса развертывания. Вы можете корректно перезапустить всех обработчиков, используя команду `queue:restart`:

```
php artisan queue:restart
```

Эта команда проинструктирует всех обработчиков очереди корректно выйти после завершения обработки своего текущего задания, чтобы существующие задания не были потеряны. Поскольку обработчики очереди выйдут при выполнении команды `queue:restart`, вы должны запустить диспетчер процессов, такой как [Supervisor](#), для автоматического перезапуска обработчиков очереди.

Очередь использует [кеш](#) для хранения сигналов перезапуска, поэтому перед использованием этой функции необходимо убедиться, что драйвер кеша правильно настроен для приложения.

## Истечение срока и тайм-ауты задания

### Истечение срока задания

В вашем файле конфигурации `config/queue.php` каждое соединение с очередью определяет параметр `retry_after`. Этот параметр указывает, сколько секунд соединение очереди должно ждать перед повторной попыткой выполнения задания, которое обрабатывается. Например, если значение `retry_after` установлено на `90`, задание будет возвращено в очередь, если оно обрабатывалось в течение 90 секунд, но не было высвобождено или удалено. Как правило, вы должны установить значение `retry_after` на максимальное количество секунд, которое может потребоваться вашим заданиям для завершения обработки.

Единственное соединение очереди, которое не содержит значения `retry_after` – это Amazon SQS. SQS будет повторять выполнение задания в соответствии с [таймаутом видимости по умолчанию](#), управляемый консолью AWS.

## Тайм-ауты обработчиков

Команда `queue:work` Artisan также содержит параметр `--timeout`. По умолчанию значение `--timeout` составляет 60 секунд. Если задание обрабатывается дольше, чем количество секунд, указанное значением тайм-аута, Обработчик, выполняющий задание, завершится с ошибкой. Обычно обработчик перезапускается автоматически [диспетчером, настроенным на вашем сервере](#):

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

Параметр конфигурации `retry_after` и параметр `--timeout` Artisan отличаются, но работают вместе, чтобы гарантировать, что задания не будут потеряны и что задания будут успешно обработаны только один раз.

Значение `--timeout` всегда должно быть как минимум на несколько секунд короче, чем ваше значение конфигурации `retry_after`. Это гарантирует, что обрабатывающий замороженное задание обработчик, всегда завершает работу перед повторной попыткой выполнения задания.

Если параметр `--timeout` выше значения конфигурации `retry_after`, то ваши задания могут быть обработаны дважды.

## # Конфигурация Supervisor

В эксплуатационном окружении вам нужен способ поддерживать процессы `queue:work` в рабочем состоянии. Процесс `queue:work` может перестать работать по разным причинам, например, из-за превышения тайм-аута обработчика или выполнения команды `queue:restart`.

По этой причине вам необходимо настроить диспетчер процессов, который может определять, когда ваши процессы `queue:work` завершаются, и автоматически перезапускать их. Кроме того, диспетчеры процессов могут позволить вам указать, сколько процессов `queue:work` вы хотите запускать одновременно. Supervisor – это диспетчер процессов, обычно используемый в средах Linux, и мы обсудим, как его настроить в следующей документации.

### Установка Supervisor

Supervisor – это диспетчер процессов для операционной системы Linux, который автоматически перезапускает ваши процессы `queue:work` в случае их сбоя. Чтобы установить Supervisor в Ubuntu, вы можете использовать следующую команду:

```
sudo apt-get install supervisor
```

Если настройка Supervisor и управление им самостоятельно кажется ошеломляющим, рассмотрите возможность использования [Laravel Forge](#), который автоматически установит и настроит Supervisor для ваших проектов Laravel.

### Настройка Supervisor

Файлы конфигурации Supervisor обычно хранятся в каталоге `/etc/supervisor/conf.d`.

В этом каталоге вы можете создать любое количество файлов конфигурации, которые сообщают Supervisor, как следует контролировать ваши процессы. Например, давайте создадим файл `laravel-worker.conf`, который запускает и отслеживает процессы `queue:work`:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-time=60
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

В этом примере директива `numprocs` инструктирует Supervisor запустить восемь процессов `queue:work` и отслеживать их все, автоматически перезапуская их в случае сбоя. Вы должны изменить директиву `command` конфигурации, чтобы отразить желаемое соединение с очередью и параметры обработчика.

Вы должны убедиться, что значение `stopwaitsecs` больше, чем количество секунд, затраченных на выполнение вашего самого продолжительного задания. В противном случае Supervisor может убить задание до того, как оно завершит обработку.

## Запуск Supervisor

После создания файла конфигурации вы можете обновить конфигурацию Supervisor и запустить процессы, используя следующие команды:

```
sudo supervisorctl reread
```

```
sudo supervisorctl update
```

```
sudo supervisorctl start "laravel-worker:*
```

Для получения дополнительной информации о Supervisor обратитесь к [документации Supervisor](#).

## # Разбор неудачных заданий

Иногда ваши задания в очереди терпят неудачу. Не волнуйтесь, не всегда все идет по плану! Laravel включает удобный способ [указать максимальное количество попыток выполнения задания](#). После того, как асинхронное задание превысит это количество попыток, оно будет вставлено в таблицу базы данных `failed_jobs`. [Синхронно отправленные задания](#), которые потерпели неудачу, не сохраняются в этой таблице, и их исключения немедленно обрабатываются приложением.

Миграция для создания таблицы `failed_jobs` обычно уже присутствует в новых приложениях Laravel. Однако, если ваше приложение не содержит миграции для этой таблицы, вы можете использовать команду `make:queue-failed-table` для создания миграции:

```
php artisan make:queue-failed-table
```

```
php artisan migrate
```

При запуске [обработчика очереди](#) вы можете указать максимальное количество попыток выполнения задания, используя переключатель `--tries` команды `queue:work`. Если вы не укажете значение для параметра `--tries`, задания будут выполняться только один раз или столько раз, сколько указано в свойстве класса задания `$tries`:

```
php artisan queue:work redis --tries=3
```

Используя параметр `--backoff`, вы можете указать, сколько секунд Laravel должен ждать перед повторной попыткой выполнения задания, для которого возникло исключение. По умолчанию задание сразу же возвращается в очередь, чтобы его можно было повторить:

```
php artisan queue:work redis --tries=3 --backoff=3
```

Если вы хотите настроить, сколько секунд Laravel должен ждать перед повторной попыткой выполнения каждого из заданий, для которого возникло исключение, вы можете сделать это, определив свойство `$backoff` в своем классе задания:

```
/**  
 * Количество секунд ожидания перед повторной попыткой выполнения задания.  
 *  
 * @var int  
 */  
public $backoff = 3;
```

Если вам требуется более сложная логика для определения времени отсрочки выполнения задания, вы можете определить метод `backoff` для своего класса задания:

```
/**  
 * Рассчитать количество секунд ожидания перед повторной попыткой выполнения задания.  
 */  
public function backoff(): int  
{  
    return 3;  
}
```

Вы можете легко настроить «экспоненциальную» отсрочку, возвращая массив значений отсрочки из метода `backoff`. В этом примере задержка повторной попытки выполнения будет составлять 1 секунду для первой попытки, 5 секунд для второй попытки и 10 секунд для третьей попытки, и 10 секунд для каждой последующей повторной попытки, если осталось еще попыток:

```
/**  
 * Рассчитать количество секунд ожидания перед повторной попыткой выполнения задания.  
 *  
 * @return array<int, int>  
 */  
public function backoff(): array  
{
```

```
    return [1, 5, 10];  
}
```

## Очистка после неудачных заданий

В случае сбоя определенного задания вы можете отправить предупреждение своим пользователям или отменить любые действия, которые были частично выполнены заданием. Для этого вы можете определить метод `failed` в своем классе работы. Экземпляр `Throwable`, который привел к сбою задания, будет передан методу `failed`:

```
<?php  
  
namespace App\Jobs;  
  
use App\Models\Podcast;  
use App\Services\AudioProcessor;  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Foundation\Queue\Queueable;  
use Throwable;  
  
class ProcessPodcast implements ShouldQueue  
{  
    use Queueable;  
  
    /**  
     * Создать новый экземпляр задания.  
     */  
    public function __construct(  
        public Podcast $podcast,  
    ) {}  
  
    /**  
     * Выполнить задание.  
     */  
    public function handle(AudioProcessor $processor): void  
    {  
        // Process uploaded podcast...  
    }  
  
    /**  
     * Обработать провал задания.  
     */  
    public function failed(?Throwable $exception): void  
    {
```

```
// Отправляем пользователю уведомление об ошибке и т.д.  
}  
}
```

Перед вызовом метода `failed` создается новый экземпляр задания. Поэтому все изменения свойств класса, которые могли произойти в методе `handle`, будут потеряны.

## Повторная попытка выполнения неудачных заданий

Чтобы просмотреть все неудачные задания, которые были вставлены в вашу таблицу базы данных `failed_jobs`, вы можете использовать команду `queue:failed` Artisan:

```
php artisan queue:failed
```

Команда `queue:failed` перечислит идентификатор задания, соединение, очередь, время сбоя и другую информацию о задании. Идентификатор задания может быть использован для повторной попытки выполнить неудачное задание. Например, чтобы повторить неудачное задание с идентификатором `ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece`, введите следующую команду:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
```

При необходимости вы можете передать команде несколько идентификаторов:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece 91401d2c-0784-4f43-824c-
```

Вы также можете повторить все неудачные задания для определенной очереди:

```
php artisan queue:retry --queue=name
```

Чтобы повторить все неудачные задания, выполните команду `queue:retry` и передайте `all` вместо идентификаторов:

```
php artisan queue:retry all
```

Если вы хотите удалить неудачные задание, вы можете использовать команду `queue:forget`:

```
php artisan queue:forget 91401d2c-0784-4f43-824c-34f94a33c24d
```

При использовании [Horizon](#) вы должны использовать команду `horizon:forget` для удаления неудачного задания вместо команды `queue:forget`.

Чтобы удалить все неудачные задания из таблицы `failed_jobs`, вы можете использовать команду `queue:flush`:

```
php artisan queue:flush
```

## Игнорирование отсутствующих моделей

При внедрении модели Eloquent в задание, модель автоматически сериализуется перед помещением в очередь и повторно извлекается из базы данных при обработке задания. Однако, если модель была удалена в то время, когда задание ожидало обработки, ваше задание может завершиться ошибкой с [ModelNotFoundException](#).

Для удобства вы можете выбрать автоматическое удаление заданий с отсутствующими моделями, установив для свойства задания `$deleteWhenMissingModels` значение `true`. Когда для этого свойства установлено значение `true`, Laravel отбрасывает задание, не вызывая исключения:

```
/**  
 * Удалить задание, если модели больше не существуют.  
 */
```

```
*  
* @var bool  
*/  
public $deleteWhenMissingModels = true;
```

## Удаление неудачных заданий

Вы можете удалить записи в таблице `failed_jobs` вашего приложения, вызвав команду `queue:prune-failed` Artisan:

```
php artisan queue:prune-failed
```

По умолчанию все записи о неудачных заданиях старше 24 часов будут удалены. Если в команде указать параметр `--hours`, будут сохранены только те записи о неудачных заданиях, которые были вставлены в течение последних N часов. Например, следующая команда удалит все записи неудачных заданий, которые были вставлены более 48 часов назад:

```
php artisan queue:prune-failed --hours=48
```

## Хранение неудачных заданий в DynamoDB

Laravel поддерживает хранение записей о неудачных заданиях в [DynamoDB](#) вместо таблицы реляционной базы данных. Перед этим вы должны вручную создать таблицу DynamoDB для хранения всех записей о неудачных заданиях. Обычно эта таблица называется `failed_jobs`, но вы должны назвать ее в зависимости от значения параметра конфигурации `queue.failed.table` в конфигурационном файле `queue` вашего приложения.

Таблица `failed_jobs` должна иметь строковый первичный partition key с именем `application` и строковый первичный sort key с именем `uuid`. Часть ключа `application` будет содержать имя вашего приложения, определенное значением конфигурации `name` в конфигурационном файле `app` вашего приложения. Поскольку имя приложения является частью ключа таблицы DynamoDB, вы можете использовать одну и ту же таблицу для хранения неудачных заданий для нескольких приложений Laravel.

Кроме того, убедитесь, что вы установили AWS SDK, чтобы ваше приложение Laravel могло работать с Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Затем установите значение параметра конфигурации `queue.failed.driver` на `dynamodb`. Кроме того, вы должны определить опции конфигурации `key`, `secret` и `region` в массиве конфигурации неудачного задания. Эти параметры будут использоваться для аутентификации в AWS. При использовании драйвера `dynamodb` опция конфигурации `queue.failed.database` не нужна:

```
'failed' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'failed_jobs',
],
```

## Отключение хранилища неудачных заданий

Вы можете указать Laravel отбрасывать невыполненные задания без их сохранения, установив параметр конфигурации `queue.failed.driver` в значение `null`. Как правило, это можно сделать с помощью переменной окружения `QUEUE_FAILED_DRIVER`:

```
QUEUE_FAILED_DRIVER=null
```

## События неудачных заданий

Если вы хотите зарегистрировать слушатель событий, который будет вызываться при сбое задания, вы можете использовать метод `failing` фасада `Queue`. Вызывать его можно например из метода `boot` сервис-провайдера `AppServiceProvider`:

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

```
use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     *
     * @return void
     */
    public function boot(): void
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
}
```

## # Удаление заданий из очередей

При использовании [Horizon](#) вы должны использовать команду `horizon:clear` для удаления заданий из очереди вместо команды `queue:clear`.

Если вы хотите удалить все задания, принадлежащие соединению и очереди по умолчанию, вы можете сделать это с помощью команды `queue:clear` Artisan:

```
php artisan queue:clear
```

Вы также можете указать аргумент `connection` и параметр `queue` для удаления заданий из конкретного соединения / очереди:

```
php artisan queue:clear redis --queue=emails
```

Удаление заданий из очередей доступно только для драйверов очереди SQS, Redis и базы данных.

Кроме того, процесс удаления в SQS занимает до 60 секунд, поэтому задания, отправленные в очередь SQS в течение 60 секунд после очистки очереди, также могут быть удалены.

## # Мониторинг очередей

Если ваша очередь получает внезапный приток заданий, она может стать перегруженной, что приведет к длительному ожиданию завершения заданий. При желании Laravel может предупредить вас, когда количество заданий в очереди превысит заданный порог.

Для этого добавьте в [планировщик](#) команду `queue:monitor` на запуск раз в минуту. Команда принимает имена очередей, которые вы хотите контролировать, а также желаемый порог количества заданий:

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

Когда команда обнаруживает очередь, количество заданий в которой превышает указанный порог, будет отправлено событие `Illuminate\Queue\Events\QueueBusy`. Вы можете прослушать это событие в `AppServiceProvider` вашего приложения, чтобы отправить уведомление вам или вашим коллегам:

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;
```

```
/**  
 * Bootstrap any application services.  
 */  
  
public function boot(): void  
{  
    Event::listen(function (QueueBusy $event) {  
        Notification::route('mail', 'dev@example.com')  
            ->notify(new QueueHasLongWaitTime(  
                $event->connection,  
                $event->queue,  
                $event->size  
            ));  
    });  
}
```

## # Тестирование

При тестировании кода, отправляющего задание, вы можете указать Laravel не выполнять само задание, поскольку код задания можно тестировать напрямую и отдельно от остального кода, отправляющего его. Конечно, чтобы протестировать само задание, вы можете создать экземпляр задания и вызвать метод `handle` непосредственно в teste.

Вы можете использовать метод `fake` фасада `Queue`, чтобы предотвратить фактическую отправку заданий очередь. После вызова метода `fake` фасада `Queue` вы можете проверять в тестах, что приложение пыталось поместить задания в очередь:

Pest      PHPUnit

```
<?php  
  
use App\Jobs\AnotherJob;  
use App\Jobs\FinalJob;  
use App\Jobs\ShipOrder;  
use Illuminate\Support\Facades\Queue;  
  
test('orders can be shipped', function () {  
    Queue::fake();  
  
    // Perform order shipping...  
  
    // Assert that no jobs were pushed...  
    Queue::assertNothingPushed();
```

```

// Assert a job was pushed to a given queue...
Queue::assertPushedOn('queue-name', ShipOrder::class);

// Assert a job was pushed twice...
Queue::assertPushed(ShipOrder::class, 2);

// Assert a job was not pushed...
Queue::assertNotPushed(AnotherJob::class);

// Assert that a Closure was pushed to the queue...
Queue::assertClosurePushed();

// Assert the total number of jobs that were pushed...
Queue::assertCount(3);
});

```

Вы можете передать функцию-замыкание методам `assertPushed` или `assertNotPushed`, чтобы подтвердить, что задание было отправлено и прошло заданный «тест на истинность». Если было отправлено хотя бы одно задание, которое проходит заданный тест, то утверждение будет успешным:

```

Queue::assertPushed(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});

```

## Подделка определённого списка заданий

Если вам нужно имитировать только определенные задания, позволяя другим заданиям выполняться нормально, вы можете передать имена классов заданий, которые следует имитировать, методу `fake`:

Pest      PHPUnit

```

test('orders can be shipped', function () {
    Queue::fake([
        ShipOrder::class,
    ]);

    // Perform order shipping...

    // Assert a job was pushed twice...

```

```
Queue::assertPushed(ShipOrder::class, 2);  
});
```

Вы можете подделать все задания, кроме набора указанных, используя метод `except`:

```
Queue::fake()->except([  
    ShipOrder::class,  
]);
```

## Тестирование цепочки заданий

Чтобы протестировать цепочки заданий, вам нужно будет использовать возможности фасада `Bus`. Метод `assertChained` фасада `Bus` может использоваться для подтверждения того, что [цепочка заданий](#) была отправлена. Метод `assertChained` принимает массив связанных заданий в качестве первого аргумента:

```
use App\Jobs\RecordShipment;  
use App\Jobs\ShipOrder;  
use App\Jobs\UpdateInventory;  
use Illuminate\Support\Facades\Bus;  
  
Bus::fake();  
  
// ...  
  
Bus::assertChained([  
    ShipOrder::class,  
    RecordShipment::class,  
    UpdateInventory::class  
]);
```

Как вы можете видеть в приведенном выше примере, массив цепочки заданий может быть массивом имен классов заданий. Однако вы также можете предоставить массив реальных экземпляров заданий. При этом Laravel гарантирует, что экземпляры заданий относятся к одному и тому же классу и имеют одинаковые значения свойств, что и связанные задания, отправленные вашим приложением:

```
Bus::assertChained([  
    new ShipOrder,
```

```
new RecordShipment,  
new UpdateInventory,  
]);
```

Вы можете использовать метод `assertDispatchedWithoutChain`, чтобы подтвердить, что задание было отправлено без цепочки заданий:

```
Bus::assertDispatchedWithoutChain(ShipOrder::class);
```

## Модификации цепочки тестирования

Если связанное задание добавляет или добавляет задания в существующую цепочку, вы можете использовать метод задания `assertHasChain`, чтобы подтвердить, что задание имеет ожидаемую цепочку оставшихся заданий:

```
$job = new ProcessPodcast;  
  
$job->handle();  
  
$job->assertHasChain([  
    new TranscribePodcast,  
    new OptimizePodcast,  
    new ReleasePodcast,  
]);
```

Метод `assertDoesntHaveChain` может использоваться для подтверждения того, что оставшаяся цепочка задания пуста:

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

## Тестирование цепочки пакетов

Если ваша цепочка заданий содержит пакет заданий, вы можете утверждать, что связанный пакет соответствует вашим ожиданиям, вставив определение `Bus::chainedBatch` в assert цепочки:

```
use App\Jobs\ShipOrder;  
use App\Jobs\UpdateInventory;  
use Illuminate\Bus\PendingBatch;
```

```
use Illuminate\Support\Facades\Bus;

Bus::assertChained([
    new ShipOrder,
    Bus::chainedBatch(function (PendingBatch $batch) {
        return $batch->jobs->count() === 3;
    }),
    new UpdateInventory,
]);

```

## Тестирование пакетов заданий

Метод `assertBatched` фасада `Bus` может использоваться для подтверждения того, что пакет заданий был отправлен. Замыкание, данное методу `assertBatched`, получает экземпляр `Illuminate\Bus\PendingBatch`, который можно использовать для проверки заданий в пакете:

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::fake();

// ...

Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
        $batch->jobs->count() === 10;
});
```

Вы можете использовать метод `assertBatchCount`, чтобы подтвердить, что было отправлено заданное количество пакетов:

```
Bus::assertBatchCount(3);
```

Вы можете использовать `assertNothingBatched`, чтобы подтвердить, что никакие пакеты не были отправлены:

```
Bus::assertNothingBatched();
```

## Тестирования заданий / Взаимодействие пакетов

Кроме того, иногда вам может потребоваться протестировать взаимодействие отдельного задания с его базовым пакетом. Например, вам может потребоваться проверить, не отменило ли задание дальнейшую обработку своего пакета. Для этого вам необходимо назначить заданию поддельный пакет с помощью метода `withFakeBatch`. Метод `withFakeBatch` возвращает массив, содержащий экземпляр задания и поддельный пакет:

```
[$job, $batch] = (new ShipOrder)->withFakeBatch();  
  
$job->handle();  
  
$this->assertTrue($batch->cancelled());  
$this->assertEmpty($batch->added);
```

## Тестирование взаимодействия заданий и очередей

Иногда вам может потребоваться проверить, что задание в очереди освобождается обратно в очередь. Или вам может потребоваться проверить, что задание удалилось само собой. Вы можете протестировать это взаимодействие с очередью, создав экземпляр задания и вызывав метод `withFakeQueueInteractions`.

Как только взаимодействие задания с очередью будет сфальсифицировано, вы можете вызвать метод `handle` для задания. После вызова задания методы `assertReleased`, `assertDeleted`, `assertNotDeleted`, `assertFailed` и `assertNotFailed` могут использоваться для создания утверждений относительно взаимодействия задания с очередью:

```
use App\Jobs\ProcessPodcast;  
  
$job = (new ProcessPodcast)->withFakeQueueInteractions();  
  
$job->handle();  
  
$job->assertReleased(delay: 30);  
$job->assertDeleted();  
$job->assertNotDeleted();  
$job->assertFailed();  
$job->assertNotFailed();
```

## # События заданий

Используя методы `before` и `after` фасада `Queue`, вы можете указать функции, которые будут выполняться до или после обработки задания в очереди. Эти функции – прекрасная возможность для дополнительной регистрации или увеличения счётчиков для панели мониторинга. Как правило, вызов этих методов осуществляется в методе `boot` сервис-провайдера. Например, мы можем использовать `AppServiceProvider`, который включен в Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }
}
```

Используя метод `looping` фасада `Queue`, вы можете указать замыкания, которые выполняются до того, как обработчик попытается получить задание из очереди. Например, вы можете зарегистрировать замыкание для отката любых транзакций, оставшихся открытыми из-за ранее неудачного задания:

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Queue;

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

# Ограничение скорости

## # Введение

# Конфигурация кеша

## # Базовое использование

# Ручное увеличение числа попыток

# Очистка счетчика попыток

## # Введение

Laravel включает простую в использовании абстракцию ограничения скорости, которая в сочетании с [кешем](#) вашего приложения обеспечивает простой способ ограничить любое действие в течение указанного периода времени.

Если вас интересует ограничение скорости входящих HTTP-запросов, обратитесь к документации [посредника для ограничения частоты запросов](#).

## Конфигурация кеша

Обычно ограничитель скорости использует кеш вашего приложения по умолчанию, как определено ключом `default` в файле конфигурации `cache` вашего приложения. Однако вы можете указать, какой драйвер кеша должен использовать ограничитель скорости, задав ключ `limiter` в файле конфигурации `cache` вашего приложения:

```
'default' => env('CACHE_STORE', 'database'),  
  
'limiter' => 'redis',
```

## # Базовое использование

Фасад `Illuminate\Support\Facades\RateLimiter` может использоваться для взаимодействия с ограничителем скорости. Самый простой метод, предлагаемый ограничителем скорости – это метод `attempt` который ограничивает скорость данного обратного вызова на заданное количество секунд.

Метод `attempt` возвращает `false` если для обратного вызова не осталось доступных попыток; в противном случае метод `attempt` вернет результат обратного вызова или `true`. Первым аргументом, принимаемым методом `attempt` является «ключ» ограничителя скорости, который может быть любой строкой по вашему выбору, представляющей действие с ограничением скорости:

```
use Illuminate\Support\Facades\RateLimiter;

$executed = RateLimiter::attempt(
    'send-message:'.$user->id,
    $perMinute = 5,
    function() {
        // Send message...
    }
);

if (! $executed) {
    return 'Too many messages sent!';
}
```

При необходимости, вы можете добавить четвёртый аргумент к методу `attempt`, который представляет собой “скорость сброса”, или количество секунд до обновления количества доступных попыток. К примеру, мы можем изменить вышеуказанный пример так, чтобы разрешить пять попыток каждые две минуты:

```
$executed = RateLimiter::attempt(
    'send-message:'.$user->id,
    $perTwoMinutes = 5,
    function() {
        // Отправить сообщение...
    },
    $decaySeconds = 120
);
```

## Ручное увеличение числа попыток

Если вы хотите вручную взаимодействовать с ограничителем скорости, доступно множество других методов. Например, вы можете вызвать метод `tooManyAttempts`, чтобы определить, не превысил ли заданный ключ ограничителя скорости максимальное количество разрешенных попыток в минуту:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
    return 'Too many attempts!';
}

RateLimiter::increment('send-message:'.$user->id);

// Send message...
```

В качестве альтернативы вы можете использовать метод `remaining` для получения количества попыток, оставшихся для данного ключа. Если для данного ключа остались повторные попытки, вы можете вызвать метод `increment`, чтобы увеличить общее количество попыток:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::remaining('send-message:'.$user->id, $perMinute = 5)) {
    RateLimiter::increment('send-message:'.$user->id);

    // Send message...
}

RateLimiter::increment('send-message:'.$user->id);

// Send message...
```

Если вы хотите увеличить значение для определенного ключа более чем на единицу, вы можете указать желаемое число для метода `increment`:

```
RateLimiter::increment('send-message:'.$user->id, amount: 5);
```

## Определение доступности ограничителя скорости

Когда у ключа больше не осталось попыток, метод `availableIn` возвращает количество секунд, оставшихся до тех пор, пока не будут доступны новые попытки:

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
    $seconds = RateLimiter::availableIn('send-message:'.$user->id);

    return 'You may try again in '.$seconds.' seconds.';
}

RateLimiter::increment('send-message:'.$user->id);

// Send message...
```

## Очистка счетчика попыток

Вы можете сбросить количество попыток для данного ключа ограничителя скорости, используя метод `clear`. Например, вы можете сбросить количество попыток, когда данное сообщение прочитано получателем:

```
use App\Models\Message;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Отметьте сообщение как прочитанное.
 */
public function read(Message $message): Message
{
    $message->markAsRead();

    RateLimiter::clear('send-message:'.$message->user_id);

    return $message;
}
```

# Строки

# Введение

# Доступные методы

# Строки

# Строки Fluent

# Строки

# Строки Fluent

## # Введение

Laravel включает в себя различные функции для работы с строковыми значениями. Многие из этих функций используются самим фреймворком; однако, вы вольны использовать их в своих собственных приложениях, если считаете их удобными.

## # Доступные методы

### Строки

---

[class\\_basename](#)

[e](#)

[preg\\_replace\\_array](#)

[Str::after](#)

[Str::afterLast](#)

[Str::apa](#)

[Str::ascii](#)

[Str::before](#)

[Str::beforeLast](#)

[Str::between](#)

[Str::betweenFirst](#)

[Str::camel](#)

[Str::charAt](#)  
[Str::contains](#)  
[Str::containsAll](#)  
[Str::doesntContain](#)  
[Str::deduplicate](#)  
[Str::endsWith](#)  
[Str::excerpt](#)  
[Str::finish](#)  
[Str::headline](#)  
[Str::inlineMarkdown](#)  
[Str::is](#)  
[Str::isAscii](#)  
[Str::isJson](#)  
[Str::isUlid](#)  
[Str::isUrl](#)  
[Str::isUuid](#)  
[Str::kebab](#)  
[Str::lcfirst](#)  
[Str::length](#)  
[Str::limit](#)  
[Str::lower](#)  
[Str::markdown](#)  
[Str::mask](#)  
[Str::orderedUuid](#)  
[Str::padBoth](#)  
[Str::padLeft](#)  
[Str::padRight](#)  
[Str::password](#)  
[Str::plural](#)  
[Str::pluralStudy](#)  
[Str::position](#)  
[Str::random](#)  
[Str::remove](#)  
[Str::repeat](#)  
[Str::replace](#)  
[Str::replaceAll](#)

[Str::replaceFirst](#)  
[Str::replaceLast](#)  
[Str::replaceMatches](#)  
[Str::replaceStart](#)  
[Str::replaceEnd](#)  
[Str::reverse](#)  
[Str::singular](#)  
[Str::slug](#)  
[Str::snake](#)  
[Str::squish](#)  
[Str::start](#)  
[Str::startsWith](#)  
[Str::study](#)  
[Str::substr](#)  
[Str::substrCount](#)  
[Str::substrReplace](#)  
[Str::swap](#)  
[Str::take](#)  
[Str::title](#)  
[Str::toBase64](#)  
[Str::toHtmlString](#)  
[Str::ucfirst](#)  
[Str::ucsSplit](#)  
[Str::upper](#)  
[Str::ulid](#)  
[Str::unwrap](#)  
[Str::uuid](#)  
[Str::wordCount](#)  
[Str::wordWrap](#)  
[Str::words](#)  
[Str::wrap](#)  
[str](#)  
[trans](#)  
[trans\\_choice](#)

# Строки Fluent

[after](#)

[afterLast](#)

[apa](#)

[append](#)

[ascii](#)

[basename](#)

[before](#)

[beforeLast](#)

[between](#)

[betweenFirst](#)

[camel](#)

[charAt](#)

[classBasename](#)

[contains](#)

[containsAll](#)

[deduplicate](#)

[dirname](#)

[endsWith](#)

[exactly](#)

[excerpt](#)

[explode](#)

[finish](#)

[headline](#)

[inlineMarkdown](#)

[is](#)

[isAscii](#)

[isEmpty](#)

[isNotEmpty](#)

[isJson](#)

[isUlid](#)

[isUrl](#)

[isUuid](#)

[kebab](#)

[lcfirst](#)

[length](#)  
[limit](#)  
[lower](#)  
[ltrim](#)  
[markdown](#)  
[mask](#)  
[match](#)  
[matchAll](#)  
[isMatch](#)  
[newLine](#)  
[padBoth](#)  
[padLeft](#)  
[padRight](#)  
[pipe](#)  
[plural](#)  
[position](#)  
[prepend](#)  
[remove](#)  
[repeat](#)  
[replace](#)  
[replaceArray](#)  
[replaceFirst](#)  
[replaceLast](#)  
[replaceMatches](#)  
[replaceStart](#)  
[replaceEnd](#)  
[rtrim](#)  
[scan](#)  
[singular](#)  
[slug](#)  
[snake](#)  
[split](#)  
[squish](#)  
[start](#)  
[startsWith](#)  
[stripTags](#)

study  
substr  
substrReplace  
swap  
take  
tap  
test  
title  
toBase64  
trim  
ucfirst  
ucssplit  
unwrap  
upper  
when  
whenContains  
whenContainsAll  
whenEmpty  
whenNotEmpty  
whenStartsWith  
whenEndsWith  
whenExactly  
whenNotExactly  
whenIs  
whenIsAscii  
whenIsUlid  
whenIsUuid  
whenTest  
wordCount  
words  
wrap

## # Строки

--()

Функция `_` переводит переданную строку перевода или ключ перевода, используя ваши [файлы локализации](#):

```
echo _('Welcome to our application');

echo _('messages.welcome');
```

Если указанная строка перевода или ключ не существует, то функция `_` вернет переданное значение. Итак, используя приведенный выше пример, функция `_` вернет `messages.welcome`, если этот ключ перевода не существует.

## class\_basename()

Функция `class_basename` возвращает имя переданного класса с удаленным пространством имен этого класса:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

## e()

Функция `e` запускает PHP-функцию `htmlspecialchars` с параметром `double_encode`, установленным по умолчанию в `true`:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

## preg\_replace\_array()

Функция `preg_replace_array` последовательно заменяет переданный шаблон в строке, используя массив:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);
```

```
// The event will take place between 8:30 and 9:00
```

## Str::after()

Метод `Str::after` возвращает все после переданного значения в строке. Если значение не существует в строке, то будет возвращена вся строка:

```
use Illuminate\Support\Str;  
  
$slice = Str::after('This is my name', 'This is');  
  
// ' my name'
```

## Str::afterLast()

Метод `Str::afterLast` возвращает все после последнего вхождения переданного значения в строке. Если значение не существует в строке, то будет возвращена вся строка:

```
use Illuminate\Support\Str;  
  
$slice = Str::afterLast('App\Http\Controllers\Controller', '\\');  
  
// 'Controller'
```

## Str::apa()

Метод `Str::apa` преобразует заданную строку в `Title Case` в соответствии с [правилами APA](#): use `Illuminate\Support\Str`;

```
$title = Str::apa('Creating A Project');  
  
// 'Creating a Project'
```

## Str::ascii()

Метод `Str::ascii` попытается транслитерировать строку в ASCII значение:

```
use Illuminate\Support\Str;

$slice = Str::ascii('ü');

// 'u'
```

## Str::before()

Метод `Str :: before` возвращает все до переданного значения в строке:

```
use Illuminate\Support\Str;

$slice = Str::before('This is my name', 'my name');

// 'This is '
```

## Str::beforeLast()

Метод `Str::beforeLast` возвращает все до последнего вхождения переданного значения в строке:

```
use Illuminate\Support\Str;

$slice = Str::beforeLast('This is my name', 'is');

// 'This '
```

## Str::between()

Метод `Str::between` возвращает часть строки между двумя значениями:

```
use Illuminate\Support\Str;

$slice = Str::between('This is my name', 'This', 'name');

// ' is my '
```

## Str::betweenFirst()

Метод `Str::betweenFirst` возвращает наименьший возможный участок строки между двумя значениями:

```
use Illuminate\Support\Str;  
  
$slice = Str::betweenFirst('[a] bc [d]', '[', ']');  
  
// 'a'
```

## Str::camel()

Метод `Str::camel` преобразует переданную строку в `camelCase`:

```
use Illuminate\Support\Str;  
  
$converted = Str::camel('foo_bar');  
  
// 'fooBar'
```

## Str::charAt()

Метод `Str::charAt` возвращает символ по указанному индексу. Если индекс выходит за границы, возвращается значение `false`:

```
use Illuminate\Support\Str;  
  
$character = Str::charAt('This is my name.', 6);  
  
// 's'
```

## Str::chopStart()

Метод `Str::chopStart` удаляет первое вхождение данного значения, только если значение появляется в начале строки:

```
use Illuminate\Support\Str;  
  
$url = Str::chopStart('https://laravel.com', 'https://');  
  
// 'laravel.com'
```

Вы также можете передать массив в качестве второго аргумента. Если строка начинается с любого значения в массиве, это значение будет удалено из строки:

```
use Illuminate\Support\Str;  
  
$url = Str::chopStart('http://laravel.com', ['https://', 'http://']);  
  
// 'laravel.com'
```

## Str::chopEnd()

Метод `Str::chopEnd` удаляет последнее вхождение данного значения, только если значение появляется в конце строки:

```
use Illuminate\Support\Str;  
  
$url = Str::chopEnd('app/Models/Photograph.php', '.php');  
  
// 'app/Models/Photograph'
```

Вы также можете передать массив в качестве второго аргумента. Если строка заканчивается любым из значений массива, это значение будет удалено из строки:

```
use Illuminate\Support\Str;  
  
$url = Str::chopEnd('laravel.com/index.php', ['/index.html', '/index.php']);  
  
// 'laravel.com'
```

## Str::contains()

Метод `Str::contains` определяет, содержит ли переданная строка указанное значение. По умолчанию этот метод чувствителен к регистру:

```
use Illuminate\Support\Str;  
  
$contains = Str::contains('This is my name', 'my');  
  
// true
```

Вы также можете указать массив значений, чтобы определить, содержит ли переданная строка какое-либо из значений:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', ['my', 'foo']);

// true
```

Вы можете отключить чувствительность к регистру, установив для аргумента `ignoreCase` значение `true`:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'MY', ignoreCase: true);

// true
```

## Str::containsAll()

Метод `Str::containsAll` определяет, содержит ли переданная строка все значения массива:

```
use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['my', 'name']);

// true
```

Вы можете отключить чувствительность к регистру, установив для аргумента `ignoreCase` значение `true`:

```
use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['MY', 'NAME'], ignoreCase: true);

// true
```



## Str::doesntContain()

Метод `Str::doesntContain` определяет, не содержит ли данная строка заданное значение. По умолчанию этот метод чувствителен к регистру:

```
use Illuminate\Support\Str;  
  
$doesntContain = Str::doesntContain('This is name', 'my');  
  
// true
```

Вы также можете передать массив значений, чтобы определить, не содержит ли данная строка каких-либо значений в массиве:

```
use Illuminate\Support\Str;  
  
$doesntContain = Str::doesntContain('This is name', ['my', 'foo']);  
  
// true
```

Вы можете отключить чувствительность к регистру, установив для аргумента `ignoreCase` значение `true`:

```
use Illuminate\Support\Str;  
  
$doesntContain = Str::doesntContain('This is name', 'MY', ignoreCase: true);  
  
// true
```

## Str::deduplicate()

Метод `Str::deduplicate` заменяет последовательные экземпляры символа единственным экземпляром этого символа в данной строке. По умолчанию метод дедуплицирует пробелы:

```
use Illuminate\Support\Str;  
  
$result = Str::deduplicate('The    Laravel    Framework');  
  
// The Laravel Framework
```

Вы можете указать другой символ для дедупликации, передав его в качестве второго аргумента метода:

```
use Illuminate\Support\Str;

$result = Str::deduplicate('The---Laravel---Framework', '-');

// The-Laravel-Framework
```

## Str::endsWith()

Метод `Str::endsWith` определяет, заканчивается ли переданная строка указанным значением:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', 'name');

// true
```

Вы также можете указать массив значений, чтобы определить, заканчивается ли переданная строка каким-либо из значений:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', ['name', 'foo']);

// true

$result = Str::endsWith('This is my name', ['this', 'foo']);

// false
```

## Str::excerpt()

Метод `Str::excerpt` извлекает отрывок из заданной строки, соответствующий первому вхождению фразы в эту строку:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'my', [
```

```
'radius' => 3
]);
// '...is my na...'
```

Опция `radius`, по умолчанию равная `100`, позволяет определить количество символов, которые должны появиться с каждой стороны усеченной строки.

Кроме того, вы можете использовать опцию `omission`, чтобы определить строку, которая будет добавлена перед и после усеченной строки:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'name', [
    'radius' => 3,
    'omission' => '(...)'
]);
// '(...) my name'
```

## Str::finish()

Метод `Str::finish` добавляет один экземпляр указанного значения в переданную строку, если она еще не заканчивается этим значением:

```
use Illuminate\Support\Str;

$adjusted = Str::finish('this/string', '/');
// this/string/
$adjusted = Str::finish('this/string/', '/');
// this/string/
```

## Str::headline()

Метод `Str::headline` преобразует строки, разделенные регистром, дефисами или подчеркиванием, в строку, разделенную пробелами, с заглавной первой буквой каждого слова:

```
use Illuminate\Support\Str;

$headline = Str::headline('steve_jobs');

// Steve Jobs

$headline = Str::headline('EmailNotificationSent');

// Email Notification Sent
```

## Str::inlineMarkdown()

Метод `Str::inlineMarkdown` преобразует Markdown в стиле GitHub в HTML в одну строку с использованием [CommonMark](#). Однако, в отличие от метода `markdown`, он не оборачивает весь сгенерированный HTML в блочный элемент:

```
use Illuminate\Support\Str;
```

```
$html = Str::inlineMarkdown('**Laravel**');

// <strong>Laravel</strong>
```

## Безопасность в Markdown

По умолчанию Markdown позволяет использовать HTML, что может привести к уязвимостям XSS (межсайтовый скрипting), если использовать его с необработанным пользовательским вводом. Согласно [документации по безопасности CommonMark](#), вы можете использовать опцию `html_input` для экранирования или удаления сырого HTML, а также опцию `allow_unsafe_links` для указания разрешения на небезопасные ссылки. Если вам нужно разрешить некоторый сырой HTML, следует пропустить скомпилированный Markdown через сторонние библиотеки, такие как HTML Purifier:

```
use Illuminate\Support\Str;

Str::inlineMarkdown('Inject: <script>alert("Hello XSS!");</script>', [
    'html_input' => 'strip',
    'allow_unsafe_links' => false,
]);

// Inject: alert("Hello XSS!");
```

## Str::is()

Метод `Str::is` определяет, соответствует ли переданная строка указанному шаблону. Допускается использование метасимвола подстановки `*`:

```
use Illuminate\Support\Str;

$matches = Str::is('foo*', 'foobar');

// true

$matches = Str::is('baz*', 'foobar');

// false
```

## Str::isAscii()

Метод `Str::isAscii` определяет, является ли переданная строка 7-битной ASCII:

```
use Illuminate\Support\Str;

$isAscii = Str::isAscii('Taylor');

// true

$isAscii = Str::isAscii('ü');

// false
```

## Str::isJson()

Метод `Str::isJson` определяет, является ли заданная строка допустимым JSON:

```
use Illuminate\Support\Str;

$result = Str::isJson('[1,2,3]');

// true

$result = Str::isJson('{"first": "John", "last": "Doe"}');

// true
```

```
$result = Str::isJson('{first: "John", last: "Doe"}');

// false
```

## Str::isUrl()

Метод `Str::isUrl` определяет, является ли заданная строка допустимым URL:

```
use Illuminate\Support\Str;

$isUrl = Str::isUrl('http://example.com');

// true

$isUrl = Str::isUrl('laravel');

// false
```

Метод `isUrl` считает широкий спектр протоколов допустимыми. Тем не менее, вы можете указать протоколы, которые должны считаться допустимыми, передав их методу `isUrl`:

```
$isUrl = Str::isUrl('http://example.com', ['http', 'https']);
```

## Str::isUlid()

Метод `Str::isUlid` определяет, является ли заданная строка допустимым ULID:

```
use Illuminate\Support\Str;

$isUlid = Str::isUlid('01gd6r360bp37zj17nxb55yv40');

// true

$isUlid = Str::isUlid('laravel');

// false
```

## Str::isUuid()

Метод `Str::isUuid` определяет, является ли заданная строка допустимым UUID:

```
use Illuminate\Support\Str;

$isValid = Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');

// true

$isValid = Str::isUuid('laravel');

// false
```

## Str::kebab()

Метод `Str::kebab` преобразует переданную строку в `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::kebab('fooBar');

// foo-bar
```

## Str::lcfirst()

Метод `Str::lcfirst` возвращает переданную строку с первым символом в нижнем регистре:

```
use Illuminate\Support\Str;

$string = Str::lcfirst('Foo Bar');

// foo Bar
```

## Str::length()

Метод `Str::length` возвращает длину переданной строки:

```
use Illuminate\Support\Str;

$length = Str::length('Laravel');
```

```
// 7
```

## Str::limit()

Метод `Str::limit` усекает переданную строку до указанной длины:

```
use Illuminate\Support\Str;  
  
$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20);  
  
// The quick brown fox...
```

Вы также можете передать третий строковый аргумент, содержимое которого будет добавлено в конец:

```
$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20, ' (...)').  
  
// The quick brown fox (...)
```



Если вы хотите сохранить полные слова при усечении строки, вы можете использовать аргумент `preserveWords`. Если этот аргумент имеет значение `true`, строка будет обрезана до ближайшей полной границы слова:

```
$truncated = Str::limit('The quick brown fox', 12, preserveWords: true);  
  
// The quick...
```

## Str::lower()

Метод `Str::lower` преобразует переданную строку в нижний регистр:

```
use Illuminate\Support\Str;  
  
$converted = Str::lower('LARAVEL');  
  
// laravel
```

## Str::markdown()

Метод `Str::markdown` конвертирует текст с разметкой [GitHub flavored Markdown](#) в HTML:

```
use Illuminate\Support\Str;

$html = Str::markdown('# Laravel');

// <h1>Laravel</h1>

$html = Str::markdown('# Taylor <b>Otwell</b>', [
    'html_input' => 'strip',
]);

// <h1>Taylor Otwell</h1>
```

## Str::mask()

Метод `Str::mask` маскирует часть строки повторяющимся символом и может использоваться для обfuscации сегментов строк, таких как адреса электронной почты и номера телефонов:

```
use Illuminate\Support\Str;

$string = Str::mask('taylor@example.com', '*', 3);

// tay*****
```

При необходимости вы можете указать отрицательное число в качестве третьего аргумента метода `mask`, который даст указание методу начать маскировку на заданном расстоянии от конца строки:

```
$string = Str::mask('taylor@example.com', '*', -15, 3);

// tay***@example.com
```

## Str::orderedUuid()

Метод `Str::orderedUuid` генерирует UUID с «префиксом временной метки», который может быть эффективно сохранен в индексированном столбце базы данных.

Каждый UUID, созданный с помощью этого метода, будет отсортирован после UUID, ранее созданных с помощью этого метода:

```
use Illuminate\Support\Str;  
  
return (string) Str::orderedUuid();
```

## Str::padBoth()

Метод `Str::padBoth` обворачивает функцию `str_pad` PHP, заполняя обе стороны строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;  
  
$padded = Str::padBoth('James', 10, '_');  
  
// '__James__'  
  
$padded = Str::padBoth('James', 10);  
  
// ' James '
```

## Str::padLeft()

Метод `Str::padLeft` обворачивает функцию `str_pad` PHP, заполняя левую часть строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;  
  
$padded = Str::padLeft('James', 10, '-=');  
  
// '----James'  
  
$padded = Str::padLeft('James', 10);  
  
// '      James'
```

## Str::padRight()

Метод `Str::padRight` обворачивает функцию `str_pad` PHP, заполняя правую часть строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;

$padded = Str::padRight('James', 10, '-');

// 'James-----'

$padded = Str::padRight('James', 10);

// 'James      '
```

## Str::password()

Метод `Str::password` можно использовать для генерации безопасного, случайного пароля заданной длины. Пароль будет состоять из комбинации букв, цифр, символов и пробелов. По умолчанию пароли имеют длину 32 символа:

```
use Illuminate\Support\Str;
```

```
$password = Str::password();

// 'EbJo2vE-AS:U,$%_gkrV4n,q~1xy/-_4'

$password = Str::password(12);

// 'qwuar>#V|i]N'
```

## Str::plural()

Метод `Str::plural` преобразует строку единственного числа в ее форму множественного числа. Эта функция поддерживает [любые из языков, поддерживаемых плюрализатором Laravel](#): use Illuminate\Support\Str;

```
$plural = Str::plural('car');

// cars

$plural = Str::plural('child');

// children
```

Вы можете передать целое число в качестве второго аргумента метода для получения строки в единственном или множественном числе:

```
use Illuminate\Support\Str;

$plural = Str::plural('child', 2);

// children

$singular = Str::plural('child', 1);

// child
```

## Str::pluralStudy()

Метод `Str::pluralStudy` преобразует строку единственного слова, отформатированную в заглавном регистре `studly`, в форму множественного числа. Эта функция поддерживает [любой из языков, поддерживаемых плюрализатором Laravel](#):

```
use Illuminate\Support\Str;

$plural = Str::pluralStudy('VerifiedHuman');

// VerifiedHumans

$plural = Str::pluralStudy('UserFeedback');

// UserFeedback
```

Вы можете передать целое число в качестве второго аргумента метода для получения строки в единственном или множественном числе:

```
use Illuminate\Support\Str;

$plural = Str::pluralStudy('VerifiedHuman', 2);

// VerifiedHumans

$singular = Str::pluralStudy('VerifiedHuman', 1);

// VerifiedHuman
```

## Str::position()

Метод `Str::position` возвращает позицию первого вхождения подстроки в строке.

Если подстрока не существует в данной строке, возвращается значение `false`:

```
use Illuminate\Support\Str;
```

```
$position = Str::position('Hello, World!', 'Hello');
```

```
// 0
```

```
$position = Str::position('Hello, World!', 'W');
```

```
// 7
```

## Str::random()

Метод `Str::random` генерирует случайную строку указанной длины. Этот метод использует функцию `random_bytes` PHP:

```
use Illuminate\Support\Str;
```

```
$random = Str::random(40);
```

## Str::remove()

Метод `Str::remove` удаляет указанную подстроку или массив подстрок в строке:

```
use Illuminate\Support\Str;
```

```
$string = 'Peter Piper picked a peck of pickled peppers.';
```

```
$removed = Str::remove('e', $string);
```

```
// Peter Pipr pickd a pck of pickld ppprs.
```

Вы можете передать `false` в качестве третьего аргумента для игнорирования регистра удаляемых подстрок.

## Str::repeat()

Метод `Str::repeat` повторяет заданную строку:

```
use Illuminate\Support\Str;

$string = 'a';

$repeat = Str::repeat($string, 5);

// aaaaa
```

## Str::replace()

Метод `Str::replace` заменяет в строке одну подстроку другой:

```
use Illuminate\Support\Str;

$string = 'Laravel 10.x';

$replaced = Str::replace('10.x', '11.x', $string);

// Laravel 11.x
```

Метод `replace` также принимает аргумент `caseSensitive`. По умолчанию метод `replace` чувствителен к регистру:

```
Str::replace('Framework', 'Laravel', caseSensitive: false);
```

## Str::replaceArray()

Метод `Str::replaceArray` последовательно заменяет указанное значение в строке, используя массив:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

## Str::replaceFirst()

Метод `Str::replaceFirst` заменяет первое вхождение переданного значения в строке:

```
use Illuminate\Support\Str;

$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps over the lazy dog');

// a quick brown fox jumps over the lazy dog
```

## Str::replaceLast()

Метод `Str::replaceLast` заменяет последнее вхождение переданного значения в строке:

```
use Illuminate\Support\Str;

$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over the lazy dog');

// the quick brown fox jumps over a lazy dog
```

## Str::replaceMatches()

Метод `Str::replaceMatches` заменяет все части строки, соответствующие шаблону, заданной строкой замены:

```
use Illuminate\Support\Str;

$replaced = Str::replaceMatches(
    pattern: '/[^A-Za-z0-9]+/',
    replace: '',
    subject: '(+1) 501-555-1000'
)

// '15015551000'
```

Метод `replaceMatches` также принимает замыкание, которое будет вызвано для каждой части строки, соответствующей заданному шаблону, что позволяет вам выполнять логику замены внутри замыкания и возвращать замененное значение:

```
use Illuminate\Support\Str;

$replaced = Str::replaceMatches('/\d/', function (array $matches) {
    return '[' . $matches[0] . ']';
}, '123');

// '[1][2][3]'
```

## Str::replaceStart()

Метод `Str::replaceStart` заменяет только первое вхождение заданного значения, если значение появляется в начале строки:

```
use Illuminate\Support\Str;

$replaced = Str::replaceStart('Hello', 'Laravel', 'Hello World');

// Laravel World

$replaced = Str::replaceStart('World', 'Laravel', 'Hello World');

// Hello World
```

## Str::replaceEnd()

Метод `Str::replaceEnd` заменяет только последнее вхождение заданного значения, если значение появляется в конце строки:

```
use Illuminate\Support\Str;

$replaced = Str::replaceEnd('World', 'Laravel', 'Hello World');

// Hello Laravel

$replaced = Str::replaceEnd('Hello', 'Laravel', 'Hello World');

// Hello World
```

## Str::reverse()

Метод `Str::reverse` переворачивает данную строку:

```
use Illuminate\Support\Str;

$reversed = Str::reverse('Hello World');

// dlroW olleH
```

## Str::singular()

Метод `Str::singular` преобразует строку в ее форму единственного числа. Эта функция поддерживает [любые из языков, поддерживаемых плюрализатором Laravel](#): use Illuminate\Support\Str;

```
$singular = Str::singular('cars');

// car

$singular = Str::singular('children');

// child
```

## Str::slug()

Метод `Str::slug` создает «дружественный фрагмент» URL-адреса из переданной строки:

```
use Illuminate\Support\Str;

$slug = Str::slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

## Str::snake()

Метод `Str::snake` преобразует переданную строку в `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::snake('fooBar');

// foo_bar
```

```
$converted = Str::snake('fooBar', '-');  
// foo-bar
```

## Str::squish()

Метод `Str::squish` удаляет все лишние пробелы из строки, включая лишние пробелы между словами:

```
use Illuminate\Support\Str;  
  
$string = Str::squish('    laravel    framework    ');  
  
// laravel framework
```

## Str::start()

Метод `Str::start` добавляет один экземпляр указанного значения в переданную строку, если она еще не начинается этим значением:

```
use Illuminate\Support\Str;  
  
$adjusted = Str::start('this/string', '/');  
  
// /this/string  
  
$adjusted = Str::start('/this/string', '/');  
  
// /this/string
```

## Str::startsWith()

Метод `Str::startsWith` определяет, начинается ли переданная строка с указанного значения:

```
use Illuminate\Support\Str;  
  
$result = Str::startsWith('This is my name', 'This');  
  
// true
```

Если передан массив возможных значений, метод `startsWith` вернет `true`, если строка начинается с любого из заданных значений:

```
$result = Str::startsWith('This is my name', ['This', 'That', 'There']);  
// true
```

## Str::studly()

Метод `Str::studly` преобразует переданную строку в `StudyCase`:

```
use Illuminate\Support\Str;  
  
$converted = Str::studly('foo_bar');  
  
// FooBar
```

## Str::substr()

Метод `Str::substr` возвращает часть строки, заданную параметрами «начало» и «длина»:

```
use Illuminate\Support\Str;  
  
$converted = Str::substr('The Laravel Framework', 4, 7);  
  
// Laravel
```

## Str::substrCount()

Метод `Str::substrCount` возвращает число вхождений подстроки в строку:

```
use Illuminate\Support\Str;  
  
$count = Str::substrCount('If you like ice cream, you will like snow cones.', 'like');  
// 2
```

## Str::substrReplace()

Метод `Str::substrReplace` заменяет текст в части строки, начиная с позиции, указанной третьим аргументом, и заменяет число символов, указанное четвертым аргументом. Передав `0` четвертым аргументом в метод, строка будет вставлена в указанную позицию без замены каких-либо существующих символов в строке:

```
use Illuminate\Support\Str;

$result = Str::substrReplace('1300', ':', 2);
// 13:

$result = Str::substrReplace('1300', ':', 2, 0);
// 13:00
```

## Str::swap()

Метод `Str::swap` заменяет несколько значений в заданной строке, используя функцию `strtr` PHP:

```
use Illuminate\Support\Str;

$string = Str::swap([
    'Tacos' => 'Burritos',
    'great' => 'fantastic',
], 'Tacos are great!');

// Burritos are fantastic!
```

## Str::take()

Метод `Str::take` возвращает указанное количество символов из начала строки:

```
use Illuminate\Support\Str;

$taken = Str::take('Build something amazing!', 5);

// Build
```

## Str::title()

Метод `Str::title` преобразует переданную строку в **Title Case**:

```
use Illuminate\Support\Str;

$converted = Str::title('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

## Str::toBase64() {.collection-method}

Метод `Str::toBase64` преобразует переданную строку в Base64:

```
use Illuminate\Support\Str;

$base64 = Str::toBase64('Laravel');

// TGFyYXZlbA==
```

## Str::toHtmlString()

Метод `Str::toHtmlString` преобразует экземпляр строки в экземпляр `Illuminate\Support\HtmlString`, который может отображаться в шаблонах Blade:

```
use Illuminate\Support\Str;

$htmlString = Str::of('Nuno Maduro')->toHtmlString();
```

## Str::transliterate()

Метод `Str::transliterate` попытается преобразовать данную строку в ее ближайшее представление ASCII:

```
use Illuminate\Support\Str;

$email = Str::transliterate('тест@ларавел.ком');

// 'test@laravel.com'
```

## Str::trim() {.collection-method}

Метод `Str::trim` удаляет пробелы (или другие символы) из начала и конца заданной строки. В отличие от встроенной функции PHP `trim`, метод `Str::trim` также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;  
  
$string = Str::trim(' foo bar ');  
  
// 'foo bar'
```

## Str::ltrim() {collection-method}

Метод `Str::ltrim` удаляет пробелы (или другие символы) с начала заданной строки. В отличие от встроенной функции PHP `ltrim`, метод `Str::ltrim` также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;  
  
$string = Str::ltrim(' foo bar ');  
  
// 'foo bar '
```

## Str::rtrim() {collection-method}

Метод `Str::rtrim` удаляет пробелы (или другие символы) с конца заданной строки. В отличие от встроенной функции PHP `rtrim`, метод `Str::rtrim` также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;  
  
$string = Str::rtrim(' foo bar ');  
  
// ' foo bar'
```

## Str::ucfirst()

Метод `Str::ucfirst` возвращает переданную строку с первой заглавной буквой:

```
use Illuminate\Support\Str;
```

```
$string = Str::ucfirst('foo bar');

// Foo bar
```

## Str::ucssplit()

Метод `Str::ucssplit` разделяет заданную строку на массив по символам в верхнем регистре:

```
use Illuminate\Support\Str;

$segments = Str::ucssplit('FooBar');

// [0 => 'Foo', 1 => 'Bar']
```

## Str::upper()

Метод `Str::upper` преобразует переданную строку в верхний регистр:

```
use Illuminate\Support\Str;

$string = Str::upper('laravel');

// LARAVEL
```

## Str::ulid()

Метод `Str::ulid` генерирует ULID, который является компактным, уникальным и упорядоченным по времени идентификатором:

```
use Illuminate\Support\Str;

return (string) Str::ulid();

// 01gd6r360bp37zj17nxb55yv40
```

Если вы хотите получить экземпляр даты `Illuminate\Support\Carbon`, представляющий дату и время создания заданного ULID, вы можете использовать метод `createFromId`, предоставленный интеграцией Carbon в Laravel:

```
use Illuminate\Support\Carbon;
use Illuminate\Support\Str;

$date = Carbon::createFromId((string) Str::ulid());
```

## Str::unwrap()

Метод `Str::unwrap` удаляет указанные строки из начала и конца заданной строки:

```
use Illuminate\Support\Str;

Str::unwrap('-Laravel-', '-');

// Laravel

Str::unwrap('{framework: "Laravel"}', '{', '}');

// framework: "Laravel"
```

## Str::uuid()

Метод `Str::uuid` генерирует UUID (версия 4):

```
use Illuminate\Support\Str;

return (string) Str::uuid();
```

## Str::wordCount()

Метод `Str::wordCount` возвращает число слов в строке:

```
use Illuminate\Support\Str;

Str::wordCount('Hello, world!'); // 2
```

## Str::wordWrap()

Метод `Str::wordWrap` переносит строку по заданному количеству символов:

```
use Illuminate\Support\Str;

$text = "The quick brown fox jumped over the lazy dog.";

Str::wordWrap($text, characters: 20, break: "<br />\n");

/*
The quick brown fox<br />
jumped over the lazy<br />
dog.
*/
```

## Str::words()

Метод `Str::words` ограничивает количество слов в строке. Дополнительная строка может быть передана этому методу через его третий аргумент, чтобы указать, какая строка должна быть добавлена в конец усеченной строки:

```
use Illuminate\Support\Str;

return Str::words('Perfectly balanced, as all things should be.', 3, ' >>>');

// Perfectly balanced, as >>>
```

## Str::wrap()

Метод `Str::wrap` обворачивает заданную строку дополнительной строкой или парой строк:

```
use Illuminate\Support\Str;

Str::wrap('Laravel', '');

// "Laravel"

Str::wrap('is', before: 'This ', after: ' Laravel!');

// This is Laravel!
```

## str()

Функция `str` возвращает новый экземпляр `Illuminate\Support\Stringable` для заданной строки. Эта функция эквивалентна методу `Str::of`:

```
$string = str('Taylor')->append(' Otwell');  
// 'Taylor Otwell'
```

Если функции `str` не передается аргумент, она возвращает экземпляр `Illuminate\Support\Str`:

```
$snake = str()->snake('FooBar');  
// 'foo_bar'
```

## trans()

Функция `trans` переводит переданный ключ перевода, используя ваши [файлы локализации](#):

```
echo trans('messages.welcome');
```

Если указанный ключ перевода не существует, функция `trans` вернет данный ключ. Итак, используя приведенный выше пример, функция `trans` вернет `messages.welcome`, если ключ перевода не существует.

## trans\_choice()

Функция `trans_choice` переводит заданный ключ перевода с изменением формы слова:

```
echo trans_choice('messages.notifications', $unreadCount);
```

Если указанный ключ перевода не существует, функция `trans_choice` вернет данный ключ. Итак, используя приведенный выше пример, функция `trans_choice` вернет `messages.notifications`, если ключ перевода не существует.

# # Строки Fluent

Строки Fluent обеспечивают более гибкий объектно-ориентированный интерфейс для работы со строковыми значениями, позволяя объединять несколько строковых операций вместе с использованием более удобочитаемого синтаксиса по сравнению с традиционными строковыми операциями.

## after

Метод `after` возвращает все после переданного значения в строке. Вся строка будет возвращена, если значение не существует в строке:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->after('This is');

// ' my name'
```

## afterLast

Метод `afterLast` возвращает все после последнего вхождения переданного значения в строке. Вся строка будет возвращена, если значение не существует в строке:

```
use Illuminate\Support\Str;

$slice = Str::of('App\Http\Controllers\Controller')->afterLast('\\');

// 'Controller'
```

## apa

Метод `apa` преобразует заданную строку в `Title Case` в соответствии с [правилами APA](#):

```
use Illuminate\Support\Str;

$converted = Str::of('a nice title uses the correct case')->apa();

// A Nice Title Uses the Correct Case
```

## append

Метод `append` добавляет указанные значения в строку:

```
use Illuminate\Support\Str;  
  
$string = Str::of('Taylor')->append(' Otwell');  
  
// 'Taylor Otwell'
```

## ascii

Метод `ascii` попытается транслитерировать строку в значение ASCII:

```
use Illuminate\Support\Str;  
  
$string = Str::of('ü')->ascii();  
  
// 'u'
```

## basename

Метод `basename` вернет завершающий компонент имени переданной строки:

```
use Illuminate\Support\Str;  
  
$string = Str::of('/foo/bar/baz')->basename();  
  
// 'baz'
```

При необходимости вы можете указать «расширение», которое будет удалено из завершающего компонента:

```
use Illuminate\Support\Str;  
  
$string = Str::of('/foo/bar/baz.jpg')->basename('.jpg');  
  
// 'baz'
```

## before

Метод `before` возвращает все до указанного значения в строке:

```
use Illuminate\Support\Str;  
  
$slice = Str::of('This is my name')->before('my name');  
  
// 'This is '
```

## beforeLast

Метод `beforeLast` возвращает все до последнего вхождения переданного значения в строку:

```
use Illuminate\Support\Str;  
  
$slice = Str::of('This is my name')->beforeLast('is');  
  
// 'This '
```

## between

Метод `between` возвращает часть строки между двумя значениями:

```
use Illuminate\Support\Str;  
  
$converted = Str::of('This is my name')->between('This', 'name');  
  
// ' is my '
```

## betweenFirst

Метод `betweenFirst` возвращает наименьший возможный участок строки между двумя значениями:

```
use Illuminate\Support\Str;  
  
$converted = Str::of('[a] bc [d]')->betweenFirst('[', ']');
```

```
// 'a'
```

## camel

Метод `camel` преобразует переданную строку в `camelCase`:

```
use Illuminate\Support\Str;  
  
$converted = Str::of('foo_bar')->camel();  
  
// 'fooBar'
```

## charAt

Метод `charAt` возвращает символ по указанному индексу. Если индекс выходит за границы, возвращается значение `false`:

```
use Illuminate\Support\Str;  
  
$character = Str::of('This is my name.')->charAt(6);  
  
// 's'
```

## classBasename

Метод `classBasename` возвращает имя класса без пространства имен:

```
use Illuminate\Support\Str;  
  
$class = Str::of('Foo\Bar\Baz')->classBasename();  
  
// 'Baz'
```

## chopStart

Метод `chopStart` удаляет первое вхождение данного значения, только если значение появляется в начале строки:

```
use Illuminate\Support\Str;
```

```
$url = Str::of('https://laravel.com')->chopStart('https://');  
// 'laravel.com'
```

Вы также можете передать массив. Если строка начинается с любого значения в массиве, это значение будет удалено из строки:

```
use Illuminate\Support\Str;  
  
$url = Str::of('http://laravel.com')->chopStart(['https://', 'http://']);  
  
// 'laravel.com'
```

## chopEnd

Метод `chopEnd` удаляет последнее вхождение данного значения, только если значение появляется в конце строки:

```
use Illuminate\Support\Str;  
  
$url = Str::of('https://laravel.com')->chopEnd('.com');  
  
// 'https://laravel'
```

Вы также можете передать массив. Если строка заканчивается любым из значений массива, это значение будет удалено из строки:

```
use Illuminate\Support\Str;  
  
$url = Str::of('http://laravel.com')->chopEnd(['.com', '.io']);  
  
// 'http://laravel'
```

## contains

Метод `contains` определяет, содержит ли переданная строка указанное значение. По умолчанию этот метод чувствителен к регистру:

```
use Illuminate\Support\Str;
```

```
$contains = Str::of('This is my name')->contains('my');

// true
```

Вы также можете указать массив значений, чтобы определить, содержит ли переданная строка какое-либо из этих значений:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains(['my', 'foo']);

// true
```

Вы можете отключить чувствительность к регистру, установив для аргумента `ignoreCase` значение `true`:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains('MY', ignoreCase: true);

// true
```

## containsAll

Метод `containsAll` определяет, содержит ли переданная строка все значения массива:

```
use Illuminate\Support\Str;

$containsAll = Str::of('This is my name')->containsAll(['my', 'name']);

// true
```

Вы можете отключить чувствительность к регистру, установив для аргумента `ignoreCase` значение `true`:

```
use Illuminate\Support\Str;

$containsAll = Str::of('This is my name')->containsAll(['MY', 'NAME'], ignoreCase: t
```

```
// true
```

## deduplicate

Метод `deduplicate` заменяет последовательные экземпляры символа единственным экземпляром этого символа в данной строке. По умолчанию метод дедуплицирует пробелы:

```
use Illuminate\Support\Str;  
  
$result = Str::of('The    Laravel    Framework')->deduplicate();  
  
// The Laravel Framework
```

Вы можете указать другой символ для дедупликации, передав его в качестве второго аргумента метода:

```
use Illuminate\Support\Str;  
  
$result = Str::of('The---Laravel---Framework')->deduplicate('-');  
  
// The-Laravel-Framework
```

## dirname

Метод `dirname` возвращает родительскую часть директории переданной строки:

```
use Illuminate\Support\Str;  
  
$string = Str::of('/foo/bar/baz')->dirname();  
  
// '/foo/bar'
```

При желании вы можете указать, сколько уровней каталогов вы хотите вырезать из строки:

```
use Illuminate\Support\Str;
```

```
$string = Str::of('/foo/bar/baz')->dirname(2);  
// '/foo'
```

## endsWith

Метод `endsWith` определяет, заканчивается ли переданная строка указанным значением:

```
use Illuminate\Support\Str;  
  
$result = Str::of('This is my name')->endsWith('name');  
  
// true
```

Вы также можете указать массив значений, чтобы определить, заканчивается ли переданная строка каким-либо из указанных значений:

```
use Illuminate\Support\Str;  
  
$result = Str::of('This is my name')->endsWith(['name', 'foo']);  
  
// true  
  
$result = Str::of('This is my name')->endsWith(['this', 'foo']);  
  
// false
```

## exactly

Метод `exactly` определяет, является ли переданная строка точным совпадением с другой строкой:

```
use Illuminate\Support\Str;  
  
$result = Str::of('Laravel')->exactly('Laravel');  
  
// true
```

## excerpt

Метод `excerpt` извлекает отрывок из заданной строки, соответствующий первому вхождению фразы в эту строку:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'my', [
    'radius' => 3
]);

// '...is my na...'
```

Опция `radius`, по умолчанию равная `100`, позволяет определить количество символов, которые должны появиться с каждой стороны усеченной строки.

Кроме того, вы можете использовать опцию `omission`, чтобы определить строку, которая будет добавлена перед и после усеченной строки:

```
use Illuminate\Support\Str;

$excerpt = Str::excerpt('This is my name', 'name', [
    'radius' => 3,
    'omission' => '(...)'
]);

// '(...) my name'
```

## explode

Метод `explode` разделяет строку по заданному разделителю и возвращает коллекцию, содержащую каждый раздел строки разбиения:

```
use Illuminate\Support\Str;

$collection = Str::of('foo bar baz')->explode(' ');

// collect(['foo', 'bar', 'baz'])
```

## finish

Метод `finish` добавляет один экземпляр указанного значения в переданную строку, если она еще не заканчивается этим значением:

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->finish('/');

// this/string

$adjusted = Str::of('this/string/')->finish('/');

// this/string/
```

## headline

Метод `headline` преобразует строки, разделенные регистром, дефисами или подчеркиваниями, в строку с пробелами, где первая буква каждого слова написана заглавной:

```
use Illuminate\Support\Str;

$headline = Str::of('taylor_otwell')->headline();

// Taylor Otwell

$headline = Str::of('EmailNotificationSent')->headline();

// Email Notification Sent
```

## inlineMarkdown

Метод `inlineMarkdown` преобразует Markdown в стиле GitHub в HTML в одну строку с использованием [CommonMark](#). Однако, в отличие от метода `markdown`, он не оборачивает весь сгенерированный HTML в блочный элемент:

```
use Illuminate\Support\Str;

$html = Str::of('**Laravel**')->inlineMarkdown();

// <strong>Laravel</strong>
```

## is

Метод `is` определяет, соответствует ли переданная строка указанному шаблону. Допускается использование метасимвола подстановки `*`:

```
use Illuminate\Support\Str;

$matches = Str::of('foobar')->is('foo*');

// true

$matches = Str::of('foobar')->is('baz*');

// false
```

## isAscii

Метод `isAscii` определяет, является ли переданная строка строкой ASCII:

```
use Illuminate\Support\Str;

$result = Str::of('Taylor')->isAscii();

// true

$result = Str::of('ü')->isAscii();

// false
```

## isEmpty

Метод `isEmpty` определяет, является ли переданная строка пустой:

```
use Illuminate\Support\Str;

$result = Str::of(' ')->trim()->isEmpty();

// true

$result = Str::of('Laravel')->trim()->isEmpty();

// false
```

## isNotEmpty

Метод `isNotEmpty` определяет, является ли переданная строка пустой:

```
use Illuminate\Support\Str;

$result = Str::of(' ')->trim()->isNotEmpty();

// false

$result = Str::of('Laravel')->trim()->isNotEmpty();

// true
```

## isJson

Метод `isJson` определяет, является ли заданная строка допустимым JSON:

```
use Illuminate\Support\Str;

$result = Str::of('[1,2,3]')->isJson();

// true

$result = Str::of('{"first": "John", "last": "Doe"}')->isJson();

// true

$result = Str::of('{first: "John", last: "Doe"}')->isJson();

// false
```

## isUlid

Метод `isUlid` определяет, является ли заданная строка ULID:

```
use Illuminate\Support\Str;

$result = Str::of('01gd6r360bp37zj17nxb55yv40')->isUlid();

// true

$result = Str::of('Taylor')->isUlid();
```

```
// false
```

## isUrl

Метод `isUrl` определяет, является ли заданная строка URL:

```
use Illuminate\Support\Str;  
  
$result = Str::of('http://example.com')->isUrl();  
  
// true  
  
$result = Str::of('Taylor')->isUrl();  
  
// false
```

Метод `isUrl` считает широкий спектр протоколов допустимыми. Тем не менее, вы можете указать протоколы, которые должны считаться допустимыми, передав их методу `isUrl`:

```
$result = Str::of('http://example.com')->isUrl(['http', 'https']);
```

## isUuid

Метод `isUuid` определяет, является ли заданная строка UUID:

```
use Illuminate\Support\Str;  
  
$result = Str::of('5ace9ab9-e9cf-4ec6-a19d-5881212a452c')->isUuid();  
  
// true  
  
$result = Str::of('Taylor')->isUuid();  
  
// false
```

## kebab

Метод `kebab` преобразует переданную строку в `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->kebab();

// foo-bar
```

## lcfirst

Метод `lcfirst` возвращает заданную строку с первым символом в нижнем регистре:

```
use Illuminate\Support\Str;

$string = Str::of('Foo Bar')->lcfirst();

// foo Bar
```

## length

Метод `length` возвращает длину переданной строки:

```
use Illuminate\Support\Str;

$length = Str::of('Laravel')->length();

// 7
```

## limit

Метод `limit` усекает переданную строку до указанной длины:

```
use Illuminate\Support\Str;

$truncated = Str::of('The quick brown fox jumps over the lazy dog')->limit(20);

// The quick brown fox...
```

Вы также можете передать второй строковый аргумент, содержимое которого будет добавлено в конец:

```
$truncated = Str::of('The quick brown fox jumps over the lazy dog')->limit(20, ' (...)')
```

Если вы хотите сохранить полные слова при усечении строки, вы можете использовать аргумент `preserveWords`. Если этот аргумент имеет значение `true`, строка будет обрезана до ближайшей полной границы слова:

```
$truncated = Str::of('The quick brown fox')->limit(12, preserveWords: true);  
// The quick...
```

## lower

Метод `lower` преобразует переданную строку в нижний регистр:

```
use Illuminate\Support\Str;  
  
$result = Str::of('LARAVEL')->lower();  
// 'laravel'
```

## markdown

Метод `markdown` преобразует Markdown в стиле GitHub в HTML:

```
use Illuminate\Support\Str;  
  
$html = Str::of('# Laravel')->markdown();  
// <h1>Laravel</h1>  
  
$html = Str::of('# Taylor <b>Otwell</b>')->markdown([  
    'html_input' => 'strip',  
]);  
// <h1>Taylor Otwell</h1>
```

## mask

Метод `mask` маскирует часть строки повторяющимся символом и может использоваться для обfuscации сегментов строк, таких как адреса электронной почты и номера телефонов:

```
use Illuminate\Support\Str;  
  
$string = Str::of('taylor@example.com')->mask('*', 3);  
  
// tay*****
```

При необходимости вы указываете отрицательное число в качестве третьего аргумента метода `mask`, который даст указание методу начать маскировку на заданном расстоянии от конца строки:

```
$string = Str::of('taylor@example.com')->mask('*', -15, 3);  
  
// tay***@example.com  
  
$string = Str::of('taylor@example.com')->mask('*', 4, -4);  
  
// tay*****.com
```

## match

Метод `match` вернет часть строки, которая соответствует указанному шаблону регулярного выражения:

```
use Illuminate\Support\Str;  
  
$result = Str::of('foo bar')->match('/bar/');  
  
// 'bar'  
  
$result = Str::of('foo bar')->match('/foo (.*)/');  
  
// 'bar'
```

## matchAll

Метод `matchAll` вернет коллекцию, содержащую части строки, которые соответствуют указанному шаблону регулярного выражения:

```
use Illuminate\Support\Str;

$result = Str::of('bar foo bar')->matchAll('/bar/');

// collect(['bar', 'bar'])
```

Если вы укажете группировку в выражении, то Laravel вернет коллекцию совпадений первой группы соответствия:

```
use Illuminate\Support\Str;

$result = Str::of('bar fun bar fly')->matchAll('/f(\w*)/');

// collect(['un', 'ly']);
```

Если совпадений не найдено, будет возвращена пустая коллекция.

## isMatch

Метод `isMatch` вернет `true`, если строка соответствует заданному регулярному выражению:

```
use Illuminate\Support\Str;

$result = Str::of('foo bar')->isMatch('/foo (.*?)');

// true

$result = Str::of('laravel')->isMatch('/foo (.*?)');

// false
```

## newLine

Метод `newLine` добавляет символ “конец строки” к строке:

```
use Illuminate\Support\Str;

$padded = Str::of('Laravel')->newLine()->append('Framework');
```

```
// 'Laravel  
// Framework'
```

## padBoth

Метод `padBoth` оборачивает функцию `str_pad` PHP, заполняя обе стороны строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;  
  
$padded = Str::of('James')->padBoth(10, '_');  
  
// '__James__'  
  
$padded = Str::of('James')->padBoth(10);  
  
// ' James '
```

## padLeft

Метод `padLeft` оборачивает функцию `str_pad` PHP, заполняя левую часть строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;  
  
$padded = Str::of('James')->padLeft(10, '-=');  
  
// '====James'  
  
$padded = Str::of('James')->padLeft(10);  
  
// ' James'
```

## padRight

Метод `padRight` оборачивает функцию `str_pad` PHP, заполняя правую часть строки другой строкой, пока конечная строка не достигнет желаемой длины:

```
use Illuminate\Support\Str;  
  
$padded = Str::of('James')->padRight(10, '-');
```

```
// 'James----'  
  
$padded = Str::of('James')->padRight(10);  
  
// 'James      '
```

## pipe

Метод `pipe` позволяет вам преобразовать строку, передав ее текущее значение указанной функции обратного вызова:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$hash = Str::of('Laravel')->pipe('md5')->prepend('Checksum: ');  
  
// 'Checksum: a5c95b86291ea299fcbe64458ed12702'  
  
$closure = Str::of('foo')->pipe(function (Stringable $str) {  
    return 'bar';  
});  
  
// 'bar'
```

## plural

Метод `plural` преобразует строку в единственном числе во множественное число. Эта функция поддерживает [любые из языков, поддерживаемых плурализатором Laravel](#): use Illuminate\Support\Str;

```
$plural = Str::of('car')->plural();  
  
// cars  
  
$plural = Str::of('child')->plural();  
  
// children
```

Вы можете передать целое число в качестве второго аргумента метода для получения строки в единственном или множественном числе:

```
use Illuminate\Support\Str;

$plural = Str::of('child')->plural(2);

// children

$plural = Str::of('child')->plural(1);

// child
```

## position

Метод `position` возвращает позицию первого вхождения подстроки в строку. Если подстрока не существует внутри строки, возвращается значение `false`:

```
use Illuminate\Support\Str;

$position = Str::of('Hello, World!')->position('Hello');

// 0

$position = Str::of('Hello, World!')->position('W');

// 7
```

## prepend

Метод `prepend` добавляет указанные значения в начало строки:

```
use Illuminate\Support\Str;

$string = Str::of('Framework')->prepend('Laravel ');

// Laravel Framework
```

## remove

Метод `remove` удаляет указанную подстроку или массив подстрок в строке:

```
use Illuminate\Support\Str;
```

```
$string = Str::of('Arkansas is quite beautiful!')->remove('quite');

// Arkansas is beautiful!
```

Вы можете передать `false` в качестве второго аргумента для игнорирования регистра удаляемых строк.

## repeat

Метод `repeat` повторяет заданную строку:

```
use Illuminate\Support\Str;

$repeated = Str::of('a')->repeat(5);

// aaaaa
```

## replace

Метод `replace` заменяет указанную строку внутри строки:

```
use Illuminate\Support\Str;

$replaced = Str::of('Laravel 6.x')->replace('6.x', '7.x');

// Laravel 7.x
```

Метод `replace` также принимает аргумент `caseSensitive`. По умолчанию метод `replace` чувствителен к регистру:

```
$replaced = Str::of('macOS 13.x')->replace(
    'macOS', 'iOS', caseSensitive: false
);
```

## replaceArray

Метод `replaceArray` последовательно заменяет указанное значение в строке, используя массив:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::of($string)->replaceArray(['?', ['8:30', '9:00']]);

// The event will take place between 8:30 and 9:00
```

## replaceFirst

Метод `replaceFirst` заменяет первое вхождение указанного значения в строке:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceFirst('the', 'a');

// a quick brown fox jumps over the lazy dog
```

## replaceLast

Метод `replaceLast` заменяет последнее вхождение указанного значения в строке:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy dog')->replaceLast('the', 'a');

// the quick brown fox jumps over a lazy dog
```

## replaceMatches

Метод `replaceMatches` заменяет все части строки, соответствующие указанному шаблону, переданной строки:

```
use Illuminate\Support\Str;

$replaced = Str::of('(+) 501-555-1000')->replaceMatches('/[^A-Za-z0-9]+/','');

// '15015551000'
```

Метод `replaceMatches` также принимает замыкание, которое будет вызвано для каждой части строки, соответствующей шаблону, что позволяет вам выполнять логику замены в замыкании и возвращать замененное значение:

```
use Illuminate\Support\Str;

$replaced = Str::of('123')->replaceMatches('/\d/', function (array $matches) {
    return '['.$matches[0].']';
});

// '[1][2][3]'
```

## replaceStart

Метод `replaceStart` заменяет только первое вхождение заданного значения, если значение появляется в начале строки:

```
use Illuminate\Support\Str;

$replaced = Str::of('Hello World')->replaceStart('Hello', 'Laravel');

// Laravel World

$replaced = Str::of('Hello World')->replaceStart('World', 'Laravel');

// Hello Laravel
```

## replaceEnd

Метод `replaceEnd` заменяет только последнее вхождение заданного значения, если значение появляется в конце строки:

```
use Illuminate\Support\Str;

$replaced = Str::of('Hello World')->replaceEnd('World', 'Laravel');

// Hello Laravel

$replaced = Str::of('Hello World')->replaceEnd('Hello', 'Laravel');

// Hello World
```

## scan

Метод `scan` анализирует входные данные из строки в коллекцию в соответствии с форматом, поддерживаемым [sscanf функцией PHP](#):

```
use Illuminate\Support\Str;  
  
$collection = Str::of('filename.jpg')->scan('%[^.].%s');  
  
// collect(['filename', 'jpg'])
```

## singular

Метод `singular` преобразует строку в ее форму единственного числа. Эта функция поддерживает [любые из языков, поддерживаемых плюрализатором Laravel](#):

```
use Illuminate\Support\Str;  
  
$singular = Str::of('cars')->singular();  
  
// car  
  
$singular = Str::of('children')->singular();  
  
// child
```

## slug

Метод `slug` создает «дружественный фрагмент» URL-адреса из переданной строки:

```
use Illuminate\Support\Str;  
  
$slug = Str::of('Laravel Framework')->slug('-');  
  
// laravel-framework
```

## snake

Метод `snake` преобразует переданную строку в `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->snake();

// foo_bar
```

## split

Метод `split` разбивает строку на коллекцию с помощью регулярного выражения:

```
use Illuminate\Support\Str;

$segments = Str::of('one, two, three')->split('/[\s,]+/');

// collect(["one", "two", "three"])
```

## squish

Метод `squish` удаляет все лишние пробелы из строки, включая лишние пробелы между словами:

```
use Illuminate\Support\Str;

$string = Str::of('    laravel    framework    ')->squish();

// laravel framework
```

## start

Метод `start` добавляет один экземпляр указанного значения в переданную строку, если она еще не начинается этим значением:

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->start('/');

// /this/string

$adjusted = Str::of('/this/string')->start('/');
```

```
// /this/string
```

## startsWith

Метод `startsWith` определяет, начинается ли переданная строка с указанного значения:

```
use Illuminate\Support\Str;  
  
$result = Str::of('This is my name')->startsWith('This');  
  
// true
```

## stripTags

Метод `stripTags` удаляет все HTML- и PHP-теги из строки:

```
use Illuminate\Support\Str;  
  
$result = Str::of('<a href="https://laravel.com">Taylor <b>Otwell</b></a>')->stripTags();  
  
// Taylor Otwell  
  
$result = Str::of('<a href="https://laravel.com">Taylor <b>Otwell</b></a>')->stripTags();  
  
// Taylor <b>Otwell</b>
```

## studly

Метод `studly` преобразует переданную строку в `StudlyCase`:

```
use Illuminate\Support\Str;  
  
$converted = Str::of('foo_bar')->studly();  
  
// FooBar
```

## substr

Метод `substr` возвращает часть строки, заданную параметрами «начало» и «длина»:

```
use Illuminate\Support\Str;

$string = Str::of('Laravel Framework')->substr(8);

// Framework

$string = Str::of('Laravel Framework')->substr(8, 5);

// Frame
```

## substrReplace

Метод `substrReplace` заменяет текст в части строки, начиная с позиции, указанной третьим аргументом, и заменяет число символов, указанное четвертым аргументом. Передав 0 четвертым аргументом в метод, строка будет вставлена в указанную позицию без замены каких-либо существующих символов в строке:

```
use Illuminate\Support\Str;

$string = Str::of('1300')->substrReplace(':', 2);

// 13:

$string = Str::of('The Framework')->substrReplace(' Laravel', 3, 0);

// The Laravel Framework
```

## swap

Метод `swap` заменяет несколько значений в строке с использованием функции `strtr` PHP:

```
use Illuminate\Support\Str;

$string = Str::of('Tacos are great!')
    ->swap([
        'Tacos' => 'Burritos',
        'great' => 'fantastic',
    ]);
```

```
// Burritos are fantastic!
```

## take

Метод `take` возвращает указанное количество символов из начала строки:

```
use Illuminate\Support\Str;  
  
$taken = Str::of('Build something amazing!')->take(5);  
  
// Build
```

## tap

Метод `tap` передает строку заданному замыканию, позволяя вам взаимодействовать с ней, не затрагивая при этом саму строку. Исходная строка возвращается методом `tap` независимо от того, что возвращает замыкание:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$string = Str::of('Laravel')  
    ->append(' Framework')  
    ->tap(function (Stringable $string) {  
        dump('String after append: '.$string);  
    })  
    ->upper();  
  
// LARAVEL FRAMEWORK
```

## test

Метод `test` определяет, соответствует ли строка переданному шаблону регулярного выражения:

```
use Illuminate\Support\Str;  
  
$result = Str::of('Laravel Framework')->test('/Laravel/');  
  
// true
```

## title

Метод `title` преобразует переданную строку в `Title Case`:

```
use Illuminate\Support\Str;  
  
$converted = Str::of('a nice title uses the correct case')->title();  
  
// A Nice Title Uses The Correct Case
```

## toBase64() {.collection-method}

Метод `toBase64` преобразует переданную строку в Base64:

```
use Illuminate\Support\Str;  
  
$base64 = Str::of('Laravel')->toBase64();  
  
// TGFyYXZl-zA==
```

## transliterate

Метод `transliterate` попытается преобразовать данную строку в ее ближайшее представление ASCII:

```
use Illuminate\Support\Str;  
  
$email = Str::of('тест@ларавел.ком')->transliterate()  
  
// 'test@laravel.com'
```

## trim

Метод `trim` обрезает переданную строку. В отличие от встроенной функции PHP `trim`, метод `trim` в Laravel также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;  
  
$string = Str::of(' Laravel ')->trim();
```

```
// 'Laravel'

$string = Str::of('/Laravel/')->trim('/');

// 'Laravel'
```

## ltrim

Метод `ltrim` обрезает левую часть строки. В отличие от встроенной функции PHP `ltrim`, метод `ltrim` в Laravel также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;

$string = Str::of(' Laravel  ')->ltrim();

// 'Laravel'

$string = Str::of('/Laravel/')->ltrim('/');

// '/Laravel'
```

## rtrim {collection-method}

Метод `rtrim` обрезает правую часть заданной строки. В отличие от встроенной функции PHP `rtrim`, метод Laravel `rtrim` также удаляет пробельные символы Юникода:

```
use Illuminate\Support\Str;

$string = Str::of(' Laravel  ')->rtrim();

// ' Laravel'

$string = Str::of('/Laravel/')->rtrim('/');

// '/Laravel'
```

## ucfirst

Метод `ucfirst` возвращает переданную строку с первой заглавной буквой:

```
use Illuminate\Support\Str;

$string = Str::of('foo bar')->ucfirst();

// Foo bar
```

## ucssplit

Метод `upper` преобразует переданную строку в верхний регистр:

```
use Illuminate\Support\Str;

$string = Str::of('Foo Bar')->ucsplit();

// collect(['Foo', 'Bar'])
```

## unwrap

Метод `unwrap` удаляет указанные строки из начала и конца заданной строки:

```
use Illuminate\Support\Str;

Str::of('-Laravel-')->unwrap('-');

// Laravel

Str::of('{framework: "Laravel"}')->unwrap('{', '}');

// framework: "Laravel"
```

## upper

Метод `upper` преобразует заданную строку в верхний регистр:

```
use Illuminate\Support\Str;

$adjusted = Str::of('laravel')->upper();

// LARAVEL
```

## when

Метод `when` вызывает указанное замыкание, если переданное условие истинно. Замыкание получит экземпляр Fluent:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$string = Str::of('Taylor')  
    ->when(true, function (Stringable $string) {  
        return $string->append(' Otwell');  
    });  
  
// 'Taylor Otwell'
```

При необходимости вы можете передать другое замыкание в качестве третьего параметра методу `when`. Это замыкание будет выполнено, если параметр условия оценивается как `false`.

## whenContains

Метод `whenContains` вызывает данное замыкание, если строка содержит заданное значение. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$string = Str::of('tony stark')  
    ->whenContains('tony', function (Stringable $string) {  
        return $string->title();  
    });  
  
// 'Tony Stark'
```

При необходимости вы можете передать другое замыкание в качестве третьего параметра метода `when`. Это замыкание будет выполнено, если строка не содержит заданного значения.

Вы также можете передать массив значений, чтобы определить, содержит ли данная строка какие-либо значения в массиве:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('tony stark')
    ->whenContains(['tony', 'hulk'], function (Stringable $string) {
        return $string->title();
});

// Tony Stark
```

## whenContainsAll

Метод `whenContainsAll` вызывает данное замыкание, если строка содержит все заданные подстроки. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('tony stark')
    ->whenContainsAll(['tony', 'stark'], function (Stringable $string) {
        return $string->title();
});

// 'Tony Stark'
```

При необходимости вы можете передать другое замыкание в качестве третьего параметра метода `when`. Это замыкание будет выполнено, если параметр условия оценивается как `false`.

## whenEmpty

Метод `whenEmpty` вызывает переданное замыкание, если строка пуста. Если замыкание возвращает значение, то это значение будет возвращено методом `whenEmpty`. Если замыкание не возвращает значение, будет возвращен экземпляр `Fluent`:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('')->whenEmpty(function (Stringable $string) {
```

```
        return $string->trim()->prepend('Laravel');
    });

// 'Laravel'
```

## whenNotEmpty

Метод `whenNotEmpty` вызывает данное замыкание, если строка не пуста. Если замыкание возвращает значение, это значение также будет возвращено методом `whenNotEmpty`. Если замыкание не возвращает значение, будет возвращен экземпляр класса `Stringable`:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('Framework')->whenNotEmpty(function (Stringable $string) {
    return $string->prepend('Laravel ');
});

// 'Laravel Framework'
```

## whenStartsWith

Метод `whenStartsWith` вызывает данное замыкание, если строка начинается с данной подстроки. Замыкание получит свободный экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('disney world')->whenStartsWith('disney', function (Stringable $st
    return $string->title();
});

// 'Disney World'
```

## whenEndsWith

Метод `whenEndsWith` вызывает данное замыкание, если строка заканчивается заданной подстрокой. Замыкание получит свободный экземпляр строки:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('disney world')->whenEndsWith('world', function (Stringable $string) {
    return $string->title();
});

// 'Disney World'
```

## whenExactly

Метод `whenExactly` вызывает данное замыкание, если строка точно соответствует заданной строке. Закрытие получит свободный экземпляр строки:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('laravel')->whenExactly('laravel', function (Stringable $string) {
    return $string->title();
});

// 'Laravel'
```

## whenNotExactly

Метод `whenExactly` вызывает данное замыкание, если строка не соответствует заданной строке. Закрытие получит свободный экземпляр строки:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('framework')->whenNotExactly('laravel', function (Stringable $string) {
    return $string->title();
});

// 'Framework'
```

## whenIs

Метод `whenIs` вызывает данное замыкание, если строка соответствует заданному шаблону. Звездочки могут использоваться в качестве подстановочных знаков. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('foo/bar')->whenIs('foo/*', function (Stringable $string) {
    return $string->append('/baz');
});

// 'foo/bar/baz'
```

## whenIsAscii

Метод `whenIsAscii` вызывает данное замыкание, если строка представляет собой 7-битный ASCII. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

$string = Str::of('laravel')->whenIsAscii(function (Stringable $string) {
    return $string->title();
});

// 'Laravel'
```

## whenIsUlid

Метод `whenIsUlid` вызывает заданное замыкание, если строка является допустимым ULID. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;

$string = Str::of('01gd6r360bp37zj17nxb55yv40')->whenIsUlid(function (Stringable $string) {
    return $string->substr(0, 8);
});

// '01gd6r36'
```



## whenIsUuid

Метод `whenIsUuid` вызывает данное замыкание, если строка является допустимым UUID. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$string = Str::of('a0a2a2d2-0b87-4a18-83f2-2529882be2de')->whenIsUuid(function (Stringable $string) {  
    return $string->substr(0, 8);  
});  
  
// 'a0a2a2d2'
```



## whenTest

Метод `whenTest` вызывает данное замыкание, если строка соответствует заданному регулярному выражению. Замыкание получит экземпляр класса `Stringable` в качестве аргумента:

```
use Illuminate\Support\Str;  
use Illuminate\Support\Stringable;  
  
$string = Str::of('laravel framework')->whenTest('/laravel/', function (Stringable $string) {  
    return $string->title();  
});  
  
// 'Laravel Framework'
```



## wordCount

Метод `wordCount` возвращает число слов в строке:

```
use Illuminate\Support\Str;  
  
Str::of('Hello, world!')->wordCount(); // 2
```

## words

Метод `words` ограничивает количество слов в строке. Дополнительная строка может быть передана этому методу, чтобы указать, какая строка должна быть добавлена в конец усеченной строки:

```
use Illuminate\Support\Str;  
  
$string = Str::of('Perfectly balanced, as all things should be.')->words(3, ' >>>');  
  
// Perfectly balanced, as >>>
```

## wrap

Метод `wrap` обворачивает данную строку дополнительной строкой или парой строк:

```
use Illuminate\Support\Str;  
  
Str::of('Laravel')->wrap('');  
  
// "Laravel"  
  
Str::is('is')->wrap(before: 'This ', after: ' Laravel!');  
  
// This is Laravel!
```

# Планирование задач

## # Введение

## # Определение расписаний

- # Планирование команд Artisan
- # Планирование отправки заданий в очереди
- # Планирование команд операционной системы
- # Параметры периодичности расписания
- # Часовые пояса
- # Предотвращение дублирования задач
- # Выполнение задач на одном сервере
- # Фоновые задачи
- # Режим технического обслуживания

## # Запуск планировщика

- # Задания с интервалом менее минуты

## # Локальный запуск планировщика

- # Результат выполнения задачи
- # Хуки выполнения задачи
- # События

## # Введение

В прошлом вы могли создавать запись конфигурации cron для каждой задачи, которую нужно было запланировать на своем сервере. Однако это может быстро стать проблемой, потому что ваше расписание задач не находится в системе управления версиями и вы должны подключаться по SSH для просмотра существующих записей cron или добавления дополнительных записей.

Планировщик команд Laravel предлагает новый подход к управлению запланированными задачами на вашем сервере. Планировщик позволяет вам быстро и выразительно определять расписание команд в самом приложении Laravel. При использовании планировщика на вашем сервере требуется только

одна запись cron. Расписание задач обычно определяется в файле `routes/console.php` вашего приложения.

## # Определение расписаний

Вы можете определить все запланированные задачи в файле `routes/console.php` вашего приложения. Для начала рассмотрим пример. В этом примере мы определим замыкание, которое будет вызываться каждый день в полночь. В замыкании мы выполним запрос к базе данных для очистки таблицы:

```
<?php

use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    DB::table('recent_users')->delete();
})->daily();
```

В дополнение к планированию с использованием замыканий вы также можете использовать [вызываемые объекты](#). Вызываемые объекты – это простые классы PHP, содержащие метод `__invoke`:

```
Schedule::call(new DeleteRecentUsers)->daily();
```

Если вы предпочитаете зарезервировать файл `routes/console.php` только для определений команд, вы можете использовать метод `withSchedule` в файле `bootstrap/app.php` вашего приложения для определения запланированных задач. Этот метод принимает замыкание, которое получает экземпляр планировщика:

```
use Illuminate\Console\Scheduling\Schedule;

->withSchedule(function (Schedule $schedule) {
    $schedule->call(new DeleteRecentUsers)->daily();
})
```

Если вы хотите просмотреть список ваших запланированных задач и их последующего запуска, то вы можете использовать команду `schedule:list` Artisan:

```
php artisan schedule:list
```

## Планирование команд Artisan

В дополнение к планированию с использованием замыканий вы также можете использовать [команды Artisan](#) и системные команды. Например, вы можете использовать метод `command` для планирования команды Artisan, используя имя команды или класс.

При планировании команд Artisan с использованием имени класса команды вы можете передать массив дополнительных аргументов командной строки, которые должны быть переданы команде при ее вызове:

```
use App\Console\Commands\SendEmailsCommand;
use Illuminate\Support\Facades\Schedule;

Schedule::command('emails:send Taylor --force')->daily();

Schedule::command(SendEmailsCommand::class, ['Taylor', '--force'])->daily();
```

## Планирование команд закрытия Artisan

Если вы хотите запланировать команду Artisan, определенную замыканием, вы можете связать методы, связанные с планированием, после определения команды:

```
Artisan::command('delete:recent-users', function () {
    DB::table('recent_users')->delete();
})->purpose('Delete recent users')->daily();
```

Если вам нужно передать аргументы команде закрытия, вы можете передать их методу `schedule`:

```
Artisan::command('emails:send {user} {--force}', function ($user) {
    // ...
})->purpose('Send emails to the specified user')->schedule(['Taylor', '--force'])->di
```

## Планирование отправки заданий в очередь

Метод `job` используется для планирования отправки [задания в очередь](#). Этот метод обеспечивает удобный способ планирования таких заданий без использования метода `call` с замыканием:

```
use App\Jobs\Heartbeat;
use Illuminate\Support\Facades\Schedule;

Schedule::job(new Heartbeat)->everyFiveMinutes();
```

Необязательные второй и третий аргументы могут быть переданы методу `job` для указания имени очереди и соединения очереди, которые должны использоваться для постановки задания в очередь:

```
use App\Jobs\Heartbeat;
use Illuminate\Support\Facades\Schedule;

// Отправляем задание в очередь «heartbeats» соединения «sqS» ...
Schedule::job(new Heartbeat, 'heartbeats', 'sqS')->everyFiveMinutes();
```

## Планирование команд операционной системы

Метод `exec` используется для передачи команды операционной системе:

```
use Illuminate\Support\Facades\Schedule;

Schedule::exec('node /home/forge/script.js')->daily();
```

## Параметры периодичности расписания

Мы уже видели несколько примеров того, как можно настроить задачу на выполнение через определенные промежутки времени. Однако существует гораздо больше параметров планирования, которые можно назначить задаче:

| Метод                                | Описание  |
|--------------------------------------|---|
| <code>-&gt;cron('* * * * *');</code> | Запустить задачу по расписанию с параметрами cron |

| <b>Метод</b>                    | <b>Описание</b>              |
|---------------------------------|------------------------------|
| ->everySecond();                | Запускать задачу ежесекундно |
| ->everyTwoSeconds();            | - каждые 2 секунды           |
| ->everyFiveSeconds();           | - каждые 5 секунд            |
| ->everyTenSeconds();            | - каждые 10 секунд           |
| ->everyFifteenSeconds();        | - каждые 15 секунд           |
| ->everyTwentySeconds();         | - каждые 20 секунд           |
| ->everyThirtySeconds();         | - каждые 30 секунд           |
| ->everyMinute();                | Запускать задачу ежеминутно  |
| ->everyTwoMinutes();            | - каждые 2 минуты            |
| ->everyThreeMinutes();          | - каждые 3 минуты            |
| ->everyFourMinutes();           | - каждые 4 минуты            |
| ->everyFiveMinutes();           | - каждые 5 минут             |
| ->everyTenMinutes();            | - каждые 10 минут            |
| ->everyFifteenMinutes();        | - каждые 15 минут            |
| ->everyThirtyMinutes();         | - каждые 30 минут            |
| ->hourly();                     | - каждый час                 |
| ->hourlyAt(17);                 | - в 17 минут каждого часа    |
| ->everyOddHour(\$minutes = 0);  | - каждый нечетный час        |
| ->everyTwoHours(\$minutes = 0); | - каждые 2 часа              |

| <b>Метод</b>                                      | <b>Описание</b>  |
|---|--|
| <code>-&gt;everyThreeHours(\$minutes = 0);</code> | - каждые 3 часа  |
| <code>-&gt;everyFourHours(\$minutes = 0);</code>  | - каждые 4 часа  |
| <code>-&gt;everySixHours(\$minutes = 0);</code>   | - каждые 6 часов   |
| <code>-&gt;daily();</code>                        | - каждый день в полночь                                  |
| <code>-&gt;dailyAt('13:00');</code>               | - ежедневно в 13:00                                      |
| <code>-&gt;twiceDaily(1, 13);</code>              | - ежедневно дважды в день: дважды в день: в 1:00 и 13:00 |
| <code>-&gt;twiceDailyAt(1, 13, 15);</code>        | - ежедневно в 1:15 и 13:15.                              |
| <code>-&gt;weekly();</code>                       | - еженедельно в воскресенье в 00:00                      |
| <code>-&gt;weeklyOn(1, '8:00');</code>            | - еженедельно в понедельник в 8:00                       |
| <code>-&gt;monthly();</code>                      | - ежемесячно первого числа в 00:00                       |
| <code>-&gt;monthlyOn(4, '15:00');</code>          | - ежемесячно 4 числа в 15:00                             |
| <code>-&gt;twiceMonthly(1, 16, '13:00');</code>   | - ежемесячно дважды в месяц: 1 и 16 числа в 13:00        |
| <code>-&gt;lastDayOfMonth('15:00');</code>        | - ежемесячно в последний день месяца в 15:00             |
| <code>-&gt;quarterly();</code>                    | - ежеквартально в первый день в 00:00                    |
| <code>-&gt;quarterlyOn(4, '14:00');</code>        | - ежеквартально в 4-й день в 14:00.                      |
| <code>-&gt;yearly();</code>                       | - ежегодно в первый день в 00:00                         |
| <code>-&gt;yearlyOn(6, 1, '17:00');</code>        | - ежегодно в июне первого числа в 17:00                  |
| <code>-&gt;timezone('America/New_York');</code>   | Установить часовой пояс для задачи                       |

Эти методы можно комбинировать с дополнительными ограничениями для создания еще более точных расписаний, которые выполняются только в определенные дни недели. Например, вы можете запланировать выполнение команды еженедельно в понедельник:

```
use Illuminate\Support\Facades\{Schedule, Weekday, Day, Month, Year};  
  
// Запускаем раз в неделю в понедельник в 13:00 ...  
Schedule::call(function () {  
    // ...  
})->weekly()->mondays()->at('13:00');  
  
// Запускаем по будням ежечасно с 8 утра до 5 вечера ...  
Schedule::command('foo')  
    ->weekdays()  
    ->hourly()  
    ->timezone('America/Chicago')  
    ->between('8:00', '17:00');
```

Список дополнительных ограничений расписания можно найти ниже:

| Метод           | Описание                                    |
|-----------------|---|
| ->weekdays();   | Ограничить выполнение задачи рабочими днями |
| ->weekends();   | – выходными днями                           |
| ->sundays();    | – воскресным днем                           |
| ->mondays();    | – понедельником                             |
| ->tuesdays();   | – вторником                                 |
| ->wednesdays(); | – средой                                    |
| ->thursdays();  | – четвергом                                 |
| ->fridays();    | – пятницей                                  |
| ->saturdays();  | – субботой                                  |

| Метод  | Описание   |
|--|--|
| <code>-&gt;days(array mixed);</code>                     | – определенными днями                                      |
| <code>-&gt;between(\$startTime, \$endTime);</code>       | – временными интервалами начала и окончания                |
| <code>-&gt;unlessBetween(\$startTime, \$endTime);</code> | – через исключение временных интервалов начала и окончания |
| <code>-&gt;when(Closure);</code>                         | – на основе истинности результата выполненного замыкания   |
| <code>-&gt;environments(\$env);</code>                   | – окружением выполнения                                    |

## Дневные ограничения

Метод `days` можно использовать для ограничения выполнения задачи определенными днями недели. Например, вы можете запланировать выполнение команды ежечасно по воскресеньям и средам:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('emails:send')
    ->hourly()
    ->days([0, 3]);
```

В качестве альтернативы вы можете использовать константы, доступные в классе `Illuminate\Console\Scheduling\Schedule`, при указании дней, в которые должна выполняться задача:

```
use Illuminate\Support\Facades;
use Illuminate\Console\Scheduling\Schedule;

Facades\Schedule::command('emails:send')
    ->hourly()
    ->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

## Ограничения с временными интервалами

Метод `between` может использоваться для ограничения выполнения задачи в зависимости от времени суток:

```
Schedule::command('emails:send')
    ->hourly()
    ->between('7:00', '22:00');
```

Точно так же метод `unlessBetween` может использоваться для исключения определенных периодов времени выполнения задачи:

```
Schedule::command('emails:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

## Условные ограничения

Метод `when` может использоваться для ограничения выполнения задачи на основе истинности результата выполненного замыкания. Другими словами, если переданное замыкание возвращает `true`, то задача будет выполняться до тех пор, пока никакие другие ограничивающие условия не препятствуют ее запуску:

```
Schedule::command('emails:send')->daily()->when(function () {
    return true;
});
```

Метод `skip` можно рассматривать как противоположный методу `when`. Если метод `skip` возвращает `true`, то запланированная задача не будет выполнена:

```
Schedule::command('emails:send')->daily()->skip(function () {
    return true;
});
```

При использовании цепочки методов `when`, запланированная команда будет выполняться только в том случае, если все условия `when` возвращают значение `true`.

## Ограничения окружения выполнения

Метод `environment` может использоваться для выполнения задач только в указанных окружениях, согласно определению [переменной APP\\_ENV окружения](#):

```
Schedule::command('emails:send')
    ->daily()
    ->environments(['staging', 'production']);
```

## Часовые пояса

Используя метод `timezone`, вы можете указать, что время запланированной задачи должно интерпретироваться в рамках переданного часового пояса:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('report:generate')
    ->timezone('America/New_York')
    ->at('2:00')
```

Если вы постоянно назначаете один и тот же часовой пояс для всех запланированных задач, то вы можете указать, какой часовой пояс должен быть назначен всем расписаниям, определив параметр `schedule_timezone` в файле конфигурации `app` вашего приложения:

```
'timezone' => env('APP_TIMEZONE', 'UTC'),
'schedule_timezone' => 'America/Chicago',
```

Помните, что в некоторых часовых поясах используется летнее время. Когда происходит переход на летнее время, ваша запланированная задача может запускаться дважды или даже не запускаться вообще. По этой причине мы рекомендуем по возможности избегать указаний часовых поясов при планировании.

## Предотвращение дублирования задач

По умолчанию запланированные задачи будут выполняться, даже если предыдущий экземпляр задачи все еще выполняется. Чтобы предотвратить это, вы можете использовать метод `withoutOverlapping`:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('emails:send')->withoutOverlapping();
```

В этом примере команда `emails:send` [Artisan](#) будет запускаться каждую минуту при условии, что она еще не запущена. Метод `withoutOverlapping` особенно полезен, если у вас есть задачи, которые разнятся по времени выполнения, что не позволяет вам точно предсказать, сколько времени займет текущая задача.

При необходимости вы можете указать, сколько минут должно пройти до окончания блокировки «перекрывающихся» задач. По умолчанию срок блокировки истекает через 24 часа:

```
Schedule::command('emails:send')->withoutOverlapping(10);
```

Внутри метод `withoutOverlapping` использует [кэш](#) вашего приложения для получения блокировок. При необходимости вы можете очистить эти блокировки, используя команду Artisan `schedule:clear-cache`. Обычно это необходимо только в случае, если задача застревает из-за непредвиденной проблемы с сервером.

## Выполнение задач на одном сервере

Чтобы использовать этот функционал, ваше приложение должно использовать по умолчанию один из следующих драйверов кеша: `database`, `memcached`, `dynamodb`, или `redis`. Кроме того, все серверы должны взаимодействовать с одним и тем же центральным сервером кеширования.

Если планировщик вашего приложения работает на нескольких серверах, то вы можете ограничить выполнение запланированного задания только на одном сервере. Например, предположим, что у вас есть запланированная задача, по которой каждую пятницу вечером создается новый отчет. Если планировщик задач работает на трех рабочих серверах, запланированная задача будет запущена на всех трех серверах и трижды сгенерирует отчет. Не очень хорошо!

Чтобы указать, что задача должна выполняться только на одном сервере, используйте метод `onOneServer` при определении запланированной задачи. Первый сервер, который получит задачу, обеспечит атомарную блокировку задания, чтобы другие серверы не могли одновременно выполнять ту же задачу:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('report:generate')
    ->fridays()
    ->at('17:00')
    ->onOneServer();
```

## Именование заданий одного сервера

Иногда вам может потребоваться запланировать отправку одного и того же задания с разными параметрами, но при этом указать Laravel запускать каждую модификацию задания на одном сервере. Для этого вы можете присвоить каждому определению расписания уникальное имя с помощью метода `name`:

```
Schedule::job(new CheckUptime('https://laravel.com'))
    ->name('check_uptime:laravel.com')
    ->everyFiveMinutes()
    ->onOneServer();

Schedule::job(new CheckUptime('https://vapor.laravel.com'))
    ->name('check_uptime:vapor.laravel.com')
    ->everyFiveMinutes()
    ->onOneServer();
```

Аналогично, для запланированных замыканий также необходимо присвоить имя, если они должны выполняться на одном сервере:

```
Schedule::call(fn () => User::resetApiRequestCount())
    ->name('reset-api-request-count')
```

```
->daily()  
->onOneServer();
```

## Фоновые задачи

По умолчанию, несколько задач, запланированных одновременно, будут выполняться последовательно в соответствии с порядком, которым они определены в вашем методе `schedule`. Если у вас есть длительные задачи, это может привести к тому, что последующие задачи начнутся намного позже, чем ожидалось. Если вы хотите запускать задачи в фоновом режиме в соответствии с планом, то вы можете использовать метод `runInBackground`:

```
use Illuminate\Support\Facades\Schedule;  
  
Schedule::command('analytics:report')  
    ->daily()  
    ->runInBackground();
```

Метод `runInBackground` может использоваться только при планировании задач с помощью методов `command` и `exec`.

## Режим технического обслуживания

Запланированные задачи вашего приложения не будут выполняться, когда приложение находится в [режиме обслуживания](#), поскольку мы не хотим, чтобы ваши задачи мешали любому незавершенному процессу обслуживания, выполняющемуся на вашем сервере. Однако, если вы хотите принудительно запустить задачу даже в режиме обслуживания, то используйте метод `evenInMaintenanceMode` при определении задачи:

```
Schedule::command('emails:send')->evenInMaintenanceMode();
```

## # Запуск планировщика

Теперь, когда мы узнали, как определять планирование задачи, давайте обсудим, как же запускать их на нашем сервере. Команда `schedule:run` Artisan проанализирует все ваши запланированные задачи и определит, нужно ли их запускать, исходя из текущего времени сервера.

Итак, при использовании планировщика Laravel нам нужно добавить только одну конфигурационную запись cron на наш сервер, которая запускает команду `schedule:run` каждую минуту. Если вы не знаете, как добавить записи cron на свой сервер, то рассмотрите возможность использования такой службы, как [Laravel Forge](#), которая может управлять записями cron за вас:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

## Задания с интервалом менее минуты

В большинстве операционных систем задания cron ограничены запуском не чаще одного раза в минуту. Тем не менее, планировщик задач Laravel позволяет вам запланировать выполнение заданий с более частыми интервалами, даже каждую секунду:

```
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    DB::table('recent_users')->delete();
})->everySecond();
```

Когда в вашем приложении определены задания с интервалом менее минуты, команда `schedule:run` будет выполняться до конца текущей минуты, а не завершится немедленно. Это позволяет команде вызывать все необходимые задания с интервалом менее минуты в течение минуты.

Поскольку задания с интервалом менее минуты, которые выполняются дольше, чем ожидалось, могут задерживать выполнение последующих заданий, рекомендуется, чтобы все такие задания были помещены в очередь заданий или выполняли команды в фоновом режиме для обработки фактической задачи:

```
use App\Jobs\DeleteRecentUsers;

Schedule::job(new DeleteRecentUsers)->everyTenSeconds();
```

```
Schedule::command('users:delete')->everyTenSeconds()->runInBackground();
```

## Прерывание задач с интервалом менее минуты:

Поскольку команда `schedule:run` выполняется в течение всей минуты при наличии задач с интервалом менее минуты, вам иногда может потребоваться прервать выполнение команды при развертывании вашего приложения. В противном случае экземпляр команды `schedule:run`, который уже выполняется, будет продолжать использовать код вашего приложения, развернутого ранее, пока не завершится текущая минута.

Для прерывания выполняющихся `schedule:run` вы можете добавить команду `schedule:interrupt` в сценарий развертывания вашего приложения. Эту команду следует вызвать после завершения развертывания вашего приложения:

```
php artisan schedule:interrupt
```

## # Локальный запуск планировщика

Как правило, на локальной машине нет необходимости в добавлении записи cron планировщика. Вместо этого вы можете использовать команду `schedule:work` Artisan. Эта команда будет работать на переднем плане и вызывать планировщик каждую минуту, пока вы не завершите команду:

```
php artisan schedule:work
```

## # Результат выполнения задачи

Планировщик Laravel предлагает несколько удобных методов для работы с выводом результатов, созданных запланированными задачами. Во-первых, используя метод `sendOutputTo`, вы можете отправить результат в файл для последующей просмотра:

```
use Illuminate\Support\Facades\{Schedule;
```

```
Schedule::command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

Если вы хотите добавить результат в указанный файл, то используйте метод `appendOutputTo`:

```
Schedule::command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

Используя метод `emailOutputTo`, вы можете отправить результат по электронной почте на любой адрес. Перед отправкой результатов выполнения задачи по электронной почте вам следует настроить [почтовые службы](#) Laravel:

```
Schedule::command('report:generate')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('taylor@example.com');
```

Если вы хотите отправить результат по электронной почте только в том случае, если запланированная (Artisan или системная) команда завершается ненулевым кодом возврата, используйте метод `emailOutputOnFailure`:

```
Schedule::command('report:generate')
    ->daily()
    ->emailOutputOnFailure('taylor@example.com');
```

Методы `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo`, and `appendOutputTo` могут использоваться только при планировании задач с помощью методов `command` и `exec`.

## # Хуки выполнения задачи

Используя методы `before` И `after`, вы можете указать замыкания, которые будут выполняться до и после выполнения запланированной задачи:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('emails:send')
    ->daily()
    ->before(function () {
        // Задача готова к выполнению ...
    })
    ->after(function () {
        // Задача выполнена ...
    });
});
```

Методы `onSuccess` И `onFailure` позволяют указать замыкания, которые будут выполняться в случае успешного или неудачного выполнения запланированной задачи. Ошибка означает, что запланированная (Artisan или системная) команда завершилась ненулевым кодом возврата:

```
Schedule::command('emails:send')
    ->daily()
    ->onSuccess(function () {
        // Задача успешно выполнена ...
    })
    ->onFailure(function () {
        // Не удалось выполнить задачу ...
    });
});
```

Если из вашей команды доступен вывод результата, то вы можете получить к нему доступ в ваших хуках `after`, `onSuccess` ИЛИ `onFailure`, указав тип экземпляра `Illuminate\Support\Stringable` в качестве аргумента `$output` замыкания при определении вашего хука:

```
use Illuminate\Support\Stringable;

Schedule::command('emails:send')
    ->daily()
    ->onSuccess(function (Stringable $output) {
        // Задача успешно выполнена ...
    })
    ->onFailure(function (Stringable $output) {
```

```
// Не удалось выполнить задачу ...  
});
```

## Пингование URL-адресов

Используя методы `pingBefore` и `thenPing`, планировщик может автоматически пинговать по-указанному URL до или после выполнения задачи. Этот метод полезен для уведомления внешней службы, такой как [Envoyer](#), о том, что ваша запланированная задача запущена или завершена:

```
Schedule::command('emails:send')  
    ->daily()  
    ->pingBefore($url)  
    ->thenPing($url);
```

Методы `pingBeforeIf` и `thenPingIf` могут использоваться для пингования по указанному URL, только если переданное условие `$condition` истинно:

```
Schedule::command('emails:send')  
    ->daily()  
    ->pingBeforeIf($condition, $url)  
    ->thenPingIf($condition, $url);
```

Методы `pingOnSuccess` и `pingOnFailure` могут использоваться для пингования по-указанному URL только в случае успешного или неудачного выполнения задачи. Ошибка означает, что запланированная (Artisan или системная) команда завершилась ненулевым кодом возврата:

```
Schedule::command('emails:send')  
    ->daily()  
    ->pingOnSuccess($successUrl)  
    ->pingOnFailure($failureUrl);
```

## # События

Laravel отправляет различные [события](#) в процессе планирования. Вы можете [определить прослушиватели](#) для любого из следующих событий:

**Наименование события**

Illuminate\Console\Events\ScheduledTaskStarting

Illuminate\Console\Events\ScheduledTaskFinished

Illuminate\Console\Events\ScheduledBackgroundTaskFinished

Illuminate\Console\Events\ScheduledTaskSkipped

Illuminate\Console\Events\ScheduledTaskFailed

# Аутентификация

## # Введение

- # Стартовые комплекты
- # Рекомендации по базе данных
- # Обзор экосистемы

## # Быстрый запуск аутентификации

- # Установка стартовых комплектов
- # Получение аутентифицированного пользователя
- # Защита маршрутов
- # Частота попыток входа в приложение

## # Самостоятельная реализация аутентификации пользователей

- # Запоминание пользователей
- # Другие методы аутентификации

## # Basic HTTP-аутентификация

- # Basic HTTP-аутентификация без сохранения состояния

## # Выход из приложения

- # Аннулирование сессий на других устройствах

## # Подтверждение пароля

- # Конфигурация подтверждения пароля
- # Маршрутизация подтверждения пароля
- # Защита маршрутов

## # Добавление своих охранников аутентификации

- # Анонимные охранники аутентификации на базе HTTP-запросов

## # Добавление своих провайдеров пользователей

- # Контракт UserProvider
- # Контракт Authenticatable

## # Автоматическое перехеширование пароля

## # События

## # Введение

Многие веб-приложения предоставляют своим пользователям возможность аутентифицироваться в приложении и «войти в систему». Реализация этого функционала в веб-приложениях может быть сложной и потенциально рискованной задачей. По этой причине Laravel стремится предоставить вам инструменты, необходимые для быстрой, безопасной и простой реализации аутентификации.

По своей сути средства аутентификации Laravel состоят из «охранников» и «провайдеров». Охранники определяют, как пользователи проходят проверку подлинности для каждого запроса. Например, Laravel поставляется с охранником [session](#), который поддерживает состояние, используя хранилище сессий и файлы Cookies.

Провайдеры определяют, как пользователи извлекаются из вашего постоянного хранилища. Laravel поставляется с поддержкой получения пользователей с помощью [Eloquent](#) и построителя запросов к базе данных. Однако вы можете определить дополнительных провайдеров, если это необходимо для вашего приложения.

Файл конфигурации аутентификации вашего приложения находится в [config/auth.php](#). Этот файл содержит несколько хорошо задокументированных вариантов для настройки поведения служб аутентификации Laravel.

Охранников и провайдеров не следует путать с «ролями» и «разрешениями». Чтобы узнать больше об авторизации действий пользователя с помощью разрешений, обратитесь к документации по [авторизации](#).

## Стартовые комплекты

Хотите быстро начать работу? Установите [стартовый комплект приложения](#) в новое приложение Laravel. После миграции базы данных перейдите в браузере по адресу [/register](#) или любому другому URL-адресу вашего приложения. Стартовые комплекты возьмут на себя создание всей вашей системы аутентификации!

**Если вы, все же решите не использовать стартовый комплект в своем текущем приложении Laravel, то установка стартового комплекта [Laravel Breeze](#) может стать прекрасной возможностью узнать, как реализуется вся функциональность аутентификации в актуальных проектах Laravel.** Поскольку Laravel Breeze создаст для вас контроллеры аутентификации, маршруты и шаблоны, то вы можете изучить код в этих файлах, чтобы узнать, как может быть реализован функционал аутентификации Laravel.

## Рекомендации по базе данных

По умолчанию Laravel содержит [модель Eloquent](#) `App\Models\User` в вашем каталоге `app/Models`. Эта модель использует по умолчанию драйвер аутентификации `eloquent`. Если ваше приложение не использует Eloquent, то вы можете использовать драйвер аутентификации `database`, основанный на построителе запросов Laravel.

При построении схемы базы данных для модели `App\Models\User` убедитесь, что длина столбца `password` не менее 60 символов. Конечно, миграция таблицы пользователей, включенная в новые приложения Laravel, уже содержит столбец, длина которого превышает эту длину.

Кроме того, вы должны убедиться, что ваша таблица `users` (или эквивалентная) содержит столбец `remember_token` с параметрами `VARCHAR(100) NULL`. Этот столбец будет использоваться для хранения токена для пользователей, которые выбирают опцию «Запомнить меня» при входе в ваше приложение. Опять же, миграция таблицы пользователей по умолчанию, которая включена в новые приложения Laravel, уже содержит этот столбец.

## Обзор экосистемы

Laravel предлагает несколько пакетов, связанных с аутентификацией. Прежде чем продолжить, мы рассмотрим общую экосистему аутентификации в Laravel и обсудим предназначение каждого пакета.

Во-первых, рассмотрим, как работает аутентификация. При использовании веб-браузера пользователь вводит свое имя пользователя и пароль через форму входа. Если эти учетные данные верны, то приложение будет хранить информацию об аутентифицированном пользователе в [сессии](#) пользователя. Файл cookie, отправленный браузеру, содержит идентификатор сессии, чтобы последующие запросы к приложению могли связать пользователя с правильной сессией. После получения файла cookie сессии, приложение извлекает данные сессии на основе

идентификатора сессии, отмечает, что аутентификационная информация была сохранена в сессии, и рассматривает пользователя как «аутентифицированного».

Когда удаленной службе необходимо пройти аутентификацию для доступа к API, файлы cookie обычно не используются для проверки подлинности, поскольку веб-браузер отсутствует. Вместо этого удаленная служба отправляет API-токен при каждом запросе к API. Приложение может проверять входящий токен по таблице допустимых API-токенов и «аутентифицировать» запрос как выполняемый пользователем, связанным с этим API-токеном.

## Службы Web-аутентификации Laravel из коробки

Laravel содержит встроенные службы аутентификации и сессии, которые обычно доступны через фасады [Auth](#) и [Session](#). Этот функционал обеспечивают аутентификацию на основе файлов Cookies для запросов, которые инициируются из веб-браузеров. Они предоставляют методы, которые позволяют вам проверять учетные данные пользователя и аутентифицировать пользователя. Кроме того, эти службы автоматически сохраняют необходимые данные аутентификации в сессии пользователя и выдают cookie сессии пользователя. В этой документации содержится информация о том, как использовать эти службы.

## Стартовые комплекты приложения

Как уже было написано в этой документации, вы можете взаимодействовать с этими службами аутентификации напрямую, чтобы создать собственный слой аутентификации вашего приложения. Однако, чтобы помочь вам быстрее приступить к работе, мы выпустили [бесплатные пакеты](#), которые обеспечивают надежную и современную основу всего слоя аутентификации. Это пакеты Laravel Breeze, Laravel Jetstream и Laravel Fortify.

[Laravel Breeze](#) – это простая, минимальная реализация всех возможностей аутентификации Laravel, включая вход в систему, регистрацию, сброс пароля, подтверждение электронной почты и подтверждение пароля. Слой представления Laravel Breeze состоит из простых [шаблонов Blade](#), стилизованных с помощью [Tailwind CSS](#). Чтобы начать использование, ознакомьтесь с документацией по [стартовым комплектам](#). Breeze также предлагает вариант создания каркасов на основе [Inertia](#) с использованием Vue или React.

[Laravel Fortify](#) – это лишь серверная часть аутентификации для Laravel, которая реализует многие возможности, описанные в этой документации, включая аутентификацию на основе файлов cookie, а также другие возможности, такие как

двуухфакторная аутентификация и проверка электронной почты. Fortify обеспечивает серверную реализацию аутентификации для Laravel Jetstream, но может использоваться и независимо в сочетании с [Laravel Sanctum](#) для обеспечения одностраничных приложений (SPA) возможностью аутентификацией с Laravel.

[Laravel Jetstream](#) – это надежный стартовый комплект, который использует и предлагает службы аутентификации Laravel Fortify, но с красивым современным пользовательским интерфейсом на основе [Tailwind CSS](#), [Livewire](#) и / или [Inertia.js](#). Laravel Jetstream дополнительно включает поддержку двухфакторной аутентификации, поддержку команды, управление сессиями браузера, управление профилями и встроенную интеграцию с [Laravel Sanctum](#) для аутентификации токена API.

## Службы API-аутентификации Laravel

Laravel предлагает два дополнительных пакета, которые помогут вам в управлении токенами API и аутентификации запросов, сделанных с помощью токенов API: [Passport](#) и [Sanctum](#). Обратите внимание, что эти библиотеки и встроенные в Laravel библиотеки аутентификации на основе файлов cookie не являются взаимоисключающими. Эти библиотеки в основном ориентированы на аутентификацию токена API, в то время как встроенные службы аутентификации ориентированы на web-аутентификацию на основе файлов cookie. Многие приложения будут использовать как встроенные службы аутентификации Laravel на основе файлов cookie, так и один из пакетов API-аутентификации Laravel.

### Passport

Passport – это провайдер аутентификации OAuth2, предлагающий различные OAuth2 Grant Types («способы запросы»), которые позволяют вам выдавать различные типы токенов. В общем, это надежный и сложный пакет для аутентификации API. Однако большинству приложений не требуются сложный функционал, предлагаемый спецификацией OAuth2, что может сбивать с толку как пользователей, так и разработчиков. Кроме того, разработчики исторически не понимали, как аутентифицировать приложения SPA или мобильные приложения с помощью провайдеров аутентификации OAuth2, таких, как Passport.

### Sanctum

В ответ на сложность OAuth2 и путаницу разработчиков мы решили создать более простой и оптимизированный пакет аутентификации, который мог бы обрабатывать

как сторонние веб-запросы из браузера, так и запросы API через токены. Эта цель была реализована с выпуском [Laravel Sanctum](#), который следует считать предпочтительным и рекомендуемым пакетом аутентификации для приложений, и который будет предлагать собственный веб-интерфейс в дополнение к API, или работать с одностраничным приложением (SPA), которое существует отдельно от серверного приложения Laravel, или приложений,лагающих мобильный клиент.

[Laravel Sanctum](#) – это гибридный пакет аутентификации через Web / API, который может управлять всем процессом аутентификации вашего приложения. Это возможно, потому что когда приложения на основе Sanctum получают запрос, Sanctum сначала определяет, содержит ли запрос файл cookie сессии, который ссылается на аутентифицированную сессию. Sanctum выполняет это, вызывая встроенные службы аутентификации Laravel, которые мы обсуждали ранее. Если запрос не аутентифицируется с помощью файла cookie сессии, то Sanctum проверит запрос на наличие токена API. Если присутствует токен API, то Sanctum аутентифицирует запрос с помощью этого токена. Чтобы узнать больше об этом процессе, обратитесь к разделу [«Как это работает»](#) документации Sanctum.

Laravel Sanctum – это пакет API, который мы выбрали для включения в стартовый комплект [Laravel Jetstream](#), потому что мы считаем, что он лучше всего подходит для большинства веб-приложений, требующих аутентификации.

## Предварительный итог и выбор вашего стека

Таким образом, если ваше приложение будет доступно через браузер, и вы создаете монолитное приложение Laravel, то ваше приложение будет использовать встроенные службы аутентификации Laravel.

Затем, если ваше приложение предлагает API, который будут использовать третьи стороны, то вы можете выбрать между [Passport](#) или [Sanctum](#), чтобы обеспечить аутентификацию токена API для вашего приложения. В целом, по возможности следует отдавать предпочтение Sanctum, поскольку это простое и полное решение для аутентификации API, аутентификации SPA и мобильной аутентификации, включая поддержку «scopes» или «abilities».

Если вы создаете одностраничное приложение (SPA), которое будет работать с серверной частью Laravel, то вам следует использовать [Laravel Sanctum](#). При использовании Sanctum вам потребуется либо [самостоятельно реализовать свои собственные маршруты аутентификации на сервере](#), либо использовать [Laravel Fortify](#) как серверную службу аутентификации, которая предлагает маршруты и

контроллеры для такого функционала, как регистрация, сброс пароля, подтверждение электронной почты и многое другое.

Passport можно выбрать, если вашему приложению необходимо абсолютно весь функционал, предоставляемый спецификацией OAuth2.

И, если вы хотите быстро начать работу, то мы рады порекомендовать пакет [Laravel Breeze](#) как быстрый способ запустить новое приложение Laravel, который уже использует предпочтительный стек аутентификации: встроенные службы аутентификации Laravel и Laravel Sanctum.

## # Быстрый запуск аутентификации

В этой части документации обсуждается аутентификация пользователей с помощью [стартовых комплектов Laravel](#), которые включают в себя каркас пользовательского интерфейса, и помогут вам быстро начать работу. Если вы хотите напрямую интегрироваться с системами аутентификации Laravel, то ознакомьтесь с документацией по [самостоятельной аутентификации пользователей](#).

## Установка стартовых комплектов

Во-первых, вы должны [установить стартовый комплект Laravel](#). Наши текущие стартовые комплекты, Laravel Breeze и Laravel Jetstream, предлагают красиво оформленные отправные точки для интеграции аутентификации в ваше новое приложение Laravel.

Laravel Breeze – это минимальная и простая реализация всех возможностей аутентификации Laravel, включая вход в систему, регистрацию, сброс пароля, подтверждение электронной почты и подтверждение пароля. Слой представления Laravel Breeze состоит из простых [шаблонов Blade](#), стилизованных с помощью [Tailwind CSS](#). Кроме того, Breeze предоставляет варианты настройки с использованием [Livewire](#) или [Inertia](#), с выбором между использованием Vue или React для создания структуры на основе Inertia.

[Laravel Jetstream](#) – это более надежный стартовый комплект для приложений, который включает поддержку построения вашего приложения с помощью [Livewire](#) или [Inertia.js](#) и [Vue](#). Кроме того, Jetstream предлагает дополнительную поддержку двухфакторной аутентификации, команд, управления профилями, управления сессиями браузера, поддержки API через [Laravel Sanctum](#), удаления аккаунтов и т. д.

## Получение аутентифицированного пользователя

После установки стартового аутентификационного комплекта и вашего разрешения пользователям регистрироваться и аутентифицироваться в приложении, вам часто будет требоваться взаимодействовать с текущим аутентифицированным пользователем. При обработке входящего запроса вы можете получить доступ к аутентифицированному пользователю с помощью метода `user` фасада `Auth`:

```
use Illuminate\Support\Facades\Auth;

// Получить текущего аутентифицированного пользователя ...
$user = Auth::user();

// Получить текущего аутентифицированного пользователя по идентификатору ...
$id = Auth::id();
```

В качестве альтернативы, как только пользователь аутентифицирован, вы можете получить доступ к аутентифицированному пользователю через экземпляр `Illuminate\Http\Request`. Помните, что объявленные типы зависимостей в методах вашего контроллера будут автоматически внедрены. Объявив объект `Illuminate\Http\Request`, вы можете получить доступ к аутентифицированному пользователю из любого метода контроллера вашего приложения с помощью метода `user` запроса:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Обновить информацию о рейсе.
     */
}
```

```
public function update(Request $request): RedirectResponse
{
    $user = $request->user();

    // ...

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

## Определение статуса аутентификации пользователя

Чтобы определить, аутентифицирован ли пользователь, выполняющий входящий HTTP-запрос, вы можете использовать метод `check` фасада `Auth`. Этот метод вернет `true`, если пользователь аутентифицирован:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // Пользователь вошел в систему...
}
```

Несмотря на то, что можно определить, аутентифицирован ли пользователь с помощью метода `check`, вы обычно будете использовать посредника для проверки статуса аутентификации пользователя перед предоставлением пользователю доступа к определенным маршрутам / контроллерам. Чтобы узнать больше об этом, ознакомьтесь с документацией по [защите маршрутов](#).

## Защита маршрутов

[Посредник маршрута](#) используется для того, чтобы разрешить только аутентифицированным пользователям доступ к указанному маршруту. Laravel содержит посредник `auth`, который представляет собой [псевдоним посредника](#) для класса `Illuminate\Auth\Middleware\Authenticate`. Поскольку этот посредник уже имеет

внутренний псевдоним в Laravel, все, что вам нужно сделать, это задать посредника к определению маршрута:

```
Route::get('/flights', function () {
    // Только аутентифицированные пользователи могут получить доступ к этому маршруту
})->middleware('auth');
```

## Перенаправление неаутентифицированных пользователей

Когда посредник `auth` обнаруживает неаутентифицированного пользователя, он перенаправляет пользователя на [именованный маршрут](#) `login`. Вы можете изменить это поведение, используя метод `redirectGuestsTo` файла `bootstrap/app.php` вашего приложения:

```
use Illuminate\Http\Request;

->withMiddleware(function (Middleware $middleware) {
    $middleware->redirectGuestsTo('/login');

    // Using a closure...
    $middleware->redirectGuestsTo(fn (Request $request) => route('login'));
})
```

## Указание охранника аутентификации

При задании посредника `auth` маршруту вы также можете указать, какой «охранник» должен использоваться для аутентификации пользователя. Указанный охранник должен соответствовать одному из указанных в массиве `guards` конфигурационного файла `config/auth.php`:

```
Route::get('/flights', function () {
    // Только аутентифицированные пользователи могут получить доступ к этому маршруту
})->middleware('auth:admin');
```

## Частота попыток входа в приложение

Если вы используете [стартовые комплекты](#) Laravel Breeze или Laravel Jetstream, то к попыткам входа в систему будет автоматически применяться ограничение. По

умолчанию, если пользователь не сможет предоставить правильные учетные данные после нескольких попыток, то он не сможет войти в систему в течение одной минуты. Частота попыток уникальна для имени пользователя / адреса электронной почты и в совокупности с IP-адресом.

Если вы хотите ограничить частоту запросов к другим маршрутам своего приложения, то ознакомьтесь с [документацией по ограничению частоты запросов](#).

## # Самостоятельная реализация аутентификации пользователей

Вам необязательно использовать каркас аутентификации, включенный в [стартовые комплекты](#) Laravel. Если вы решите не использовать их, то вам нужно будет управлять аутентификацией пользователей напрямую, используя классы аутентификации Laravel. Не волнуйтесь, это круто!

Мы получим доступ к службам аутентификации Laravel через [фасад Auth](#), поэтому нам нужно обязательно импортировать фасад `Auth` в верхней части нашего класса. Далее, давайте проверим метод `attempt`. Метод `attempt` обычно используется для обработки попыток аутентификации из формы входа в систему вашего приложения. Если аутентификация прошла успешно, то вы должны повторно создать [сессию](#) пользователя, чтобы предотвратить [фиксацию сессии](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Обработка попыток аутентификации.
     *

```

```

 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function authenticate(Request $request): RedirectResponse
{
    $credentials = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if (Auth::attempt($credentials)) {
        $request->session()->regenerate();

        return redirect()->intended('dashboard');
    }

    return back()->withErrors([
        'email' => 'The provided credentials do not match our records.',
    ])->onlyInput('email');
}

}

```

Метод `attempt` принимает массив пар ключ / значение в качестве своего первого аргумента. Значения в массиве будут использоваться для поиска пользователя в таблице базы данных. Итак, в приведенном выше примере пользователь будет извлечен по значению столбца `email`. Если пользователь найден, то хешированный пароль, хранящийся в базе данных, будет сравниваться со значением `password`, переданным в метод через массив. Вы не должны хешировать значение пароля входящего запроса, поскольку фреймворк автоматически хеширует это значение, прежде чем сравнивать его с хешированным паролем в базе данных. Если два хешированных пароля совпадают, то для пользователя будет запущена аутентифицированная сессия.

Помните, что службы аутентификации Laravel будут получать пользователей из вашей базы данных на основе конфигурации провайдера вашего охранника аутентификации. В конфигурационном файле `config/auth.php` по умолчанию указан провайдер пользователей Eloquent, и ему дано указание использовать модель `App\Models\User` при получении пользователей. Вы можете изменить эти значения в своем файле конфигурации в зависимости от потребностей вашего приложения.

Метод `attempt` вернет `true`, если аутентификация прошла успешно. В противном случае будет возвращено `false`.

Метод `intended` экземпляра `Illuminate\Routing\Redirector` Laravel, будет перенаправлять пользователя на URL-адрес, к которому он пытался получить доступ, прежде чем он будет перехвачен посредником аутентификации. Этому методу может быть предоставлен резервный URI в случае, если адрес, переданный методу `intended` недоступен.

## Указание дополнительных условий

При желании вы также можете добавить дополнительные условия запроса к запросу аутентификации в дополнение к электронной почте и паролю пользователя. Для этого мы можем просто добавить условия запроса в массив, переданному методу `attempt`. Например, мы можем проверить, что пользователь отмечен как «активный»:

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {  
    // Authentication was successful...  
}
```

Для сложных условий запроса вы можете предоставить замыкание в массив ваших учетных данных. Это замыкание будет вызвано с экземпляром запроса, позволяя вам настраивать запрос в соответствии с потребностями вашего приложения:

```
use Illuminate\Database\Eloquent\Builder;  
  
if (Auth::attempt([  
    'email' => $email,  
    'password' => $password,  
    fn (Builder $query) => $query->has('activeSubscription'),  
])) {  
    // Authentication was successful...  
}
```

В этих примерах `email` не является обязательным параметром, он просто используется в качестве примера. Вы должны использовать любое имя столбца, равнозначное «имени пользователя» в таблице базы данных.

Метод `attemptWhen`, который принимает замыкание в качестве второго аргумента, может использоваться для более тщательной проверки потенциального пользователя перед фактической аутентификацией. Замыкание получает потенциального пользователя и должно возвращать `true` или `false` для указания, может ли пользователь быть аутентифицирован:

```
if (Auth::attempt([
    'email' => $email,
    'password' => $password,
], function (User $user) {
    return $user->isNotBanned();
})) {
    // Authentication was successful...
}
```

## Доступ к конкретному экземпляру охранника аутентификации

Используя метод `guard` фасада `Auth`, вы можете указать, какой экземпляр охранника вы хотите использовать при аутентификации пользователя. Это позволяет вам управлять аутентификацией для отдельных частей вашего приложения с использованием абсолютно отдельных аутентифицируемых моделей или пользовательских таблиц.

Имя охранника, переданное методу `guard`, должно соответствовать одному из настроенных в вашем файле конфигурации `auth.php` охраннику:

```
if (Auth::guard('admin')->attempt($credentials)) {
    // ...
}
```

## Запоминание пользователей

Многие веб-приложения содержат флажок «Запомнить меня» в форме входа. Если вы хотите реализовать функционал «Запомнить меня» в своем приложении, то вы можете передать логическое значение в качестве второго аргумента метода `attempt`.

Когда это значение равно `true`, Laravel будет поддерживать аутентификацию пользователя неопределенного долго или до тех пор, пока он не выйдет из системы вручную. Ваша таблица `users` должна включать столбец `remember_token`, который будет использоваться для хранения токена функционала «Запомнить меня». Миграция таблицы пользователей, входящая в новые приложения Laravel, уже содержит этот столбец:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // Запоминаем пользователя ...
}
```

Если ваше приложение предоставляет функционал “запомнить меня”, вы можете использовать метод `viaRemember`, чтобы определить, был ли текущий аутентифицированный пользователь аутентифицирован с использованием cookie “запомнить меня”:

```
use Illuminate\Support\Facades\Auth;

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

## Другие методы аутентификации

### Аутентификация пользователя по экземпляру модели

Если вам нужно задать экземпляр существующего пользователя в качестве текущего аутентифицированного, то вы можете передать этот экземпляр методу `login` фасада `Auth`. Переданный экземпляр пользователя должен быть реализацией [контракта](#) `Illuminate\Contracts\Auth\Authenticatable`. Модель `App\Models\User`, поставляемая с Laravel, уже реализует этот интерфейс. Этот метод аутентификации полезен, когда у вас уже есть экземпляр пользователя, например, сразу после того, как пользователь регистрируется в вашем приложении:

```
use Illuminate\Support\Facades\Auth;

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

Вы можете передать логическое значение в качестве второго аргумента метода `login`. Это значение указывает, требуется ли для аутентифицированной сессии функциональность «Запомнить меня». Помните, это означает, что сессия будет аутентифицироваться бесконечно или до тех пор, пока пользователь вручную не выйдет из приложения:

```
Auth::login($user, $remember = true);
```

При необходимости вы можете указать охранника аутентификации перед вызовом метода `login`:

```
Auth::guard('admin')->login($user);
```

## Аутентификация пользователя по идентификатору

Для аутентификации пользователя с использованием первичного ключа записи в базе данных вы можете использовать метод `loginUsingId`. Этот метод принимает первичный ключ пользователя, которого вы хотите аутентифицировать:

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

Вы можете передать логическое значение в аргумент `remember` метода `loginUsingId`. Это значение указывает, требуется ли для аутентифицированной сессии функциональность «Запомнить меня». Помните, это означает, что сессия будет аутентифицироваться бесконечно или до тех пор, пока пользователь вручную не выйдет из приложения:

```
Auth::loginUsingId(1, remember: true);
```

## Аутентификация пользователя для текущего запроса

Вы можете использовать метод `once` для аутентификации пользователя в приложении только для одного запроса. При вызове этого метода не будут использоваться сессии или файлы cookie:

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

```
}
```

## # Basic HTTP-аутентификация

[Basic HTTP-аутентификация](#) обеспечивает быстрый способ аутентификации пользователей вашего приложения без создания специальной «страницы входа». Для начала задайте маршруту [посредника auth.basic](#). Посредник `auth.basic` поставляется с Laravel, поэтому вам не нужно его определять:

```
Route::get('/profile', function () {
    // Только аутентифицированные пользователи могут получить доступ к этому маршруту
})->middleware('auth.basic');
```



После того как посредник задан маршруту, вам будет автоматически предложено ввести учетные данные при доступе к маршруту в вашем браузере. По умолчанию посредник `auth.basic` предполагает, что столбец `email` в вашей таблице базы данных `users` является его «логином».

### Примечание о FastCGI

Если вы используете PHP FastCGI и Apache для своего приложения Laravel, то аутентификация HTTP Basic может работать некорректно. Чтобы исправить эти проблемы, в файл `.htaccess` вашего приложения можно добавить следующие строки:

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

## Basic HTTP-аутентификация без сохранения состояния

Вы также можете использовать Basic HTTP-аутентификацию без задания cookie идентификатора пользователя в сессии. Это в первую очередь полезно, если вы решите использовать HTTP-аутентификацию для аутентификации запросов к API вашего приложения. Для этого [определите посредника](#), который вызывает метод `onceBasic`. Если метод `onceBasic` не возвращает ответа, то запрос может быть передан дальше в приложение:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Symfony\Component\HttpFoundation\Response;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Обработка входящего запроса.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        return Auth::onceBasic() ?: $next($request);
    }
}
```

Затем присоедините middleware к маршруту:

```
Route::get('/api/user', function () {
    // Только аутентифицированные пользователи могут получить доступ к этому маршруту
})->middleware(AuthenticateOnceWithBasicAuth::class);
```

## # Выход из приложения

Чтобы обеспечить пользователю возможность выхода из вашего приложения, вы можете использовать метод `logout` фасада `Auth`. Это удалит информацию аутентификации из сессии пользователя, так что последующие запросы не будут аутентифицированы.

В дополнение к вызову метода `logout` рекомендуется аннулировать сессию пользователя и повторно сгенерировать его токен CSRF. После выхода пользователя из системы вы обычно перенаправляете пользователя в корень вашего приложения:

```
use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

/**
 * Выход пользователя из приложения.
 */
public function logout(Request $request): RedirectResponse
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

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

## Аннулирование сессий на других устройствах

Laravel также предлагает механизм для «выхода» пользователя и аннулирования сессий, активных на других устройствах, без аннулирования сессии на его текущем устройстве. Этот функционал обычно используется, когда пользователь меняет или обновляет свой пароль, и вы хотите аннулировать сессии на других устройствах, сохранив аутентификацию текущего устройства.

Перед тем как начать, вы должны убедиться, что middleware [Illuminate\Session\Middleware\AuthenticateSession](#) включено на маршрутах, которые должны использовать аутентификацию сессии. Обычно вы должны размещать это middleware в определении группы маршрутов, чтобы оно применялось к большинству маршрутов вашего приложения. По умолчанию middleware [AuthenticateSession](#) может быть присоединено к маршруту с использованием [псевдонима посредника auth.session](#):

```
Route::middleware(['auth', 'auth.session'])->group(function () {
    Route::get('/', function () {
        // ...
    });
});
```

Затем вы можете использовать метод [logoutOtherDevices](#) фасада [Auth](#). Этот метод требует, чтобы пользователь подтвердил свой текущий пароль, который ваше

приложение должно принять через форму ввода:

```
use Illuminate\Support\Facades\Auth;  
  
Auth::logoutOtherDevices($currentPassword);
```

Когда вызывается метод `logoutOtherDevices`, другие сессии пользователя будут полностью аннулированы, то есть он будет «отключен» от всех охранников, которым он ранее был аутентифицированы.

## # Подтверждение пароля

При создании приложения вы можете потребовать от пользователя подтверждения пароля перед выполнением действия или перед перенаправлением пользователя в конфиденциальный раздел приложения. Laravel содержит встроенный посредник для упрощения этого процесса. Реализация этого функционала потребует от вас определения двух маршрутов: один маршрут для отображения шаблона, предлагающего пользователю подтвердить свой пароль, и другой маршрут для уточнения действительности пароля и дальнейшего перенаправления его к необходимому разделу.

В следующей документации обсуждается, как напрямую интегрироваться с функционалом подтверждения пароля Laravel; однако, если вы хотите начать работу быстрее, то [стартовые комплекты Laravel](#) уже включают поддержку этого функционала!

## Конфигурация подтверждения пароля

После подтверждения пароля пользователю не будет предлагаться повторно подтвердить пароль в течение трех часов. Однако вы можете настроить время, по истечении которого пользователю будет предложено повторно ввести пароль, изменив значение параметра конфигурации `password_timeout` в файле конфигурации `config/auth.php` вашего приложения.

# Маршрутизация подтверждения пароля

## Форма подтверждения пароля

Сначала мы определим маршрут для отображения шаблона формы с запросом у пользователя подтверждения своего пароля:

```
Route::get('/confirm-password', function () {
    return view('auth.confirm-password');
})->middleware('auth')->name('password.confirm');
```

Как и следовало ожидать, шаблон, возвращаемый этим маршрутом, должен иметь форму, содержащую поле `password`. Кроме того, добавьте текст, который объясняет, что пользователь входит в защищенный раздел приложения и должен подтвердить свой пароль.

## Процесс подтверждения пароля

Затем мы определим маршрут, который будет обрабатывать запрос формы из шаблона «подтвердить пароль». Этот маршрут будет отвечать за проверку пароля и перенаправление пользователя к месту назначения:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1']);
```

Прежде чем двигаться дальше, давайте рассмотрим этот маршрут более подробно. Во-первых, определяется, что поле `password` запроса действительно соответствует паролю аутентифицированного пользователя. Если пароль

действителен, то нам нужно сообщить сессии Laravel, что пользователь подтвердил свой пароль. Метод `passwordConfirmed` устанавливает временную метку в сессии пользователя, которую Laravel может использовать, чтобы определить, когда пользователь последний раз подтвердил свой пароль. Наконец, мы можем перенаправить пользователя по назначению.

## Защита маршрутов

Вы должны убедиться, что любому маршруту, связанному с подтверждением пароля, назначен посредник `password.confirm`. Этот посредник входит в стандартную установку Laravel и автоматически сохраняет предполагаемое место назначения пользователя в сессии, чтобы пользователя можно было перенаправить в это место после подтверждения своего пароля. После сохранения предполагаемого пункта назначения пользователя в сессии посредник перенаправит пользователя на [именованный маршрут `password.confirm`](#):

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

## # Добавление своих охранников аутентификации

Вы можете определить своих собственных охранников аутентификации, используя метод `extend` фасада `Auth`. Вы должны разместить свой вызов метода `extend` внутри [поставщика служб](#). Поскольку Laravel уже содержит `AppServiceProvider`, мы можем разместить код в этом поставщике:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\ServiceProvider;
```

```
class AppServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Auth::extend('jwt', function (Application $app, string $name, array $config)
            // Возвращаем экземпляр `Illuminate\Contracts\Auth\Guard` ...
            return new JwtGuard(Auth::createUserProvider($config['provider']));
        );
    }
}
```

Как вы можете видеть в приведенном выше примере, замыкание, переданное методу `extend`, должно возвращать реализацию `Illuminate\Contracts\Auth\Guard`. Этот интерфейс содержит несколько методов, которые вам необходимо реализовать для определения своего охранника. После того как ваш охранник был определен, вы можете ссылаться на него в конфигурации `guards` конфигурационного файла `config/auth.php`:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

## Анонимные охранники аутентификации на базе HTTP-запросов

Самый простой способ реализовать собственную систему аутентификации на базе HTTP-запросов – использовать метод `Auth::viaRequest`. Этот метод позволяет быстро определить процесс аутентификации с помощью одного замыкания.

Для начала вызовите метод `Auth::viaRequest` в методе `boot AppServiceProvider` вашего приложения. Метод `viaRequest` принимает имя драйвера аутентификации в качестве своего первого аргумента. Это имя может быть любой строкой, описывающей вашего охранника. Второй аргумент, передаваемый методу, должно быть

замыкание, которое принимает входящий HTTP-запрос и возвращает экземпляр пользователя или, если аутентификация не удалась, то `null`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Загрузка любых служб приложения.
 */
public function boot(): void
{
    $this->registerPolicies();

    Auth::viaRequest('custom-token', function (Request $request) {
        return User::where('token', (string) $request->token)->first();
    });
}
```

После того как ваш драйвер аутентификации был определен, вы можете настроить его как драйвер в конфигурации `guards` конфигурационного файла `config/auth.php`:

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

Наконец, вы можете ссылаться на охранника при назначении middleware аутентификации для маршрута:

```
Route::middleware('auth:api')->group(function () {
    // ...
});
```

## # Добавление своих провайдеров пользователей

Если вы не используете традиционную реляционную базу данных для хранения своих пользователей, то вам нужно будет расширить Laravel своим собственным

провайдером аутентификации пользователей. Мы будем использовать метод `provider` фасада `Auth` для определения собственного провайдера пользователей. Провайдер пользователей должен вернуть реализацию `Illuminate\Contracts\Auth\UserProvider`:

```
<?php

namespace App\Providers;

use App\Extensions\MongoUserProvider;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Auth::provider('mongo', function (Application $app, array $config) {
            // Возвращаем экземпляр `Illuminate\Contracts\Auth\UserProvider` ...

            return new MongoUserProvider($app->make('mongo.connection'));
        });
    }
}
```

После того как вы зарегистрировали провайдера с помощью метода `provider`, вы можете переключиться на нового провайдера пользователей в конфигурационном файле `config/auth.php`. Сначала определите провайдера, который использует ваш новый драйвер:

```
'providers' => [
    'users' => [
        'driver' => 'mongo',
    ],
],
```

Наконец, вы можете указать этого провайдера в своей конфигурации `guards`:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

## Контракт UserProvider

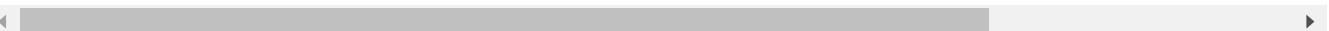
Реализации `Illuminate\Contracts\Auth\UserProvider` отвечают за получение реализации `Illuminate\Contracts\Auth\Authenticatable` из системы постоянного хранения, такой как MySQL, MongoDB и т. д. Эти два интерфейса позволяют механизмам аутентификации Laravel продолжать функционировать независимо от того, как хранятся пользовательские данные или какой тип класса используется для представления аутентифицированного пользователя.

Давайте посмотрим на контракт `Illuminate\Contracts\Auth\UserProvider`:

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider
{
    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);
    public function rehashPasswordIfRequired(Authenticatable $user, array $credentials);
}
```



- Метод `retrieveById` обычно принимает ключ, представляющий пользователя, такой как автоинкрементный идентификатор из базы данных MySQL. Реализация `Authenticatable`, соответствующая идентификатору, должна быть получена и возвращена методом.
- Метод `retrieveByToken` извлекает пользователя по его уникальному идентификатору `$identifier` и `$token`, обычно хранящимся в столбце

`remember_token` базы данных. Как и в предыдущем методе, этот метод должен вернуть реализацию `Authenticatable` соответствующую значению токена.

- Метод `updateRememberToken` обновляет `remember_token` экземпляра `$user` новым `$token`. Новый токен назначается пользователям при успешной попытке аутентификации с отмеченным флагком «Запомнить меня» или когда пользователь выходит из системы.
- Метод `retrieveByCredentials` принимает массив учетных данных, переданный методу `Auth::attempt` при попытке аутентификации в приложении. Затем метод должен «запросить» у постоянного хранилища пользователя, соответствующего этим учетным данным. Как правило, этот метод запускает запрос с условием `WHERE`, который ищет запись пользователя с «именем пользователя», равнозначным `$credentials['имя пользователя']`. Метод должен возвращать реализацию `Authenticatable`. **Этот метод не должен пытаться выполнить проверку пароля или аутентификацию.**
- Метод `validateCredentials` должен сравнивать переданный `$user` с `$credentials` для аутентификации пользователя. Например, этот метод обычно использует метод `Hash::check` для сравнения значения `$user->getAuthPassword()` со значением `$credentials['password']`. Этот метод должен возвращать `true` или `false`, указывая, действителен ли пароль.
- Метод `rehashPasswordIfRequired` должен перехешировать пароль данного пользователя `$user`, если он требуется и поддерживается. Например, этот метод обычно использует метод `Hash::needsRehash`, чтобы определить, нужно ли перехешировать значение `$credentials['password']`. Если пароль необходимо перехэшировать, метод должен использовать метод `Hash::make` для повторного хеширования пароля и обновления записи пользователя в базовом постоянном хранилище.

## Контракт `Authenticatable`

Теперь, когда мы изучили каждый из методов `UserProvider`, давайте взглянем на контракт `Authenticatable`. Помните, что провайдеры пользователей должны возвращать реализации этого интерфейса из методов `retrieveById`, `retrieveByToken`, и `retrieveByCredentials`:

```
<?php
```

```
namespace Illuminate\Contracts\Auth;
```

```
interface Authenticatable
{
    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPasswordName();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();
}
```

Этот интерфейс прост. Метод `getAuthIdentifierName` должен возвращать имя столбца «первичного ключа» пользователя, а метод `getAuthIdentifier` должен возвращать «первичный ключ» пользователя. При использовании серверной части MySQL это, вероятно, будет автоинкрементный первичный ключ, присваиваемый записи пользователя. Метод `getAuthPasswordName` должен возвращать имя столбца пароля пользователя. Метод `getAuthPassword` должен возвращать хешированный пароль пользователя.

Этот интерфейс позволяет системе аутентификации работать с любым классом `User`, независимо от того, какой ORM или уровень абстракции хранилища вы используете. По умолчанию Laravel включает класс `App\Models\User` в каталог `app/Models`, который реализует этот интерфейс.

## # Автоматическое перехеширование пароля

Алгоритм хеширования паролей в Laravel по умолчанию — `bcrypt`. «Рабочий коэффициент» для хэшей `bcrypt` можно настроить с помощью файла конфигурации вашего приложения `config/hashing.php` или переменной среды `BCRYPT_ROUNDS`.

Обычно коэффициент работы `bcrypt` следует увеличивать с течением времени по мере увеличения вычислительной мощности процессора/графического процессора. Если вы увеличите рабочий фактор `bcrypt` для своего приложения, Laravel будет корректно и автоматически перехешировать пароли пользователей, когда пользователи проходят аутентификацию в вашем приложении с помощью стартовых наборов Laravel или когда вы аутентифицируете пользователей вручную с помощью метода `attempt`.

Обычно автоматическое изменение пароля не должно нарушать работу приложения; однако вы можете отключить это поведение, опубликовав файл

конфигурации `hashing`:

```
php artisan config:publish hashing
```

После публикации файла конфигурации вы можете установить для параметра конфигурации `rehash_on_login` значение `false`:

```
'rehash_on_login' => false,
```

## # События

Laravel отправляет различные [события](#) в процессе аутентификации. Вы можете [определить слушателей](#) для любого из следующих событий:

### Наименование события

`Illuminate\Auth\Events\Registered`

`Illuminate\Auth\Events\Attempting`

`Illuminate\Auth\Events\Authenticated`

`Illuminate\Auth\Events\Login`

`Illuminate\Auth\Events\Failed`

`Illuminate\Auth\Events\Validated`

`Illuminate\Auth\Events\Verified`

`Illuminate\Auth\Events\Logout`

`Illuminate\Auth\Events\CurrentDeviceLogout`

`Illuminate\Auth\Events\OtherDeviceLogout`

`Illuminate\Auth\Events\Lockout`

**Наименование события**

Illuminate\Auth\Events\PasswordReset

# Авторизация

## # Введение

## # Шлюзы (Gate)

- # Написание шлюзов
- # Авторизация действий через шлюзы
- # Ответы шлюза
- # Хуки шлюзов
- # Встроенная авторизация

## # Создание политик

- # Генерация политик
- # Регистрация политик

## # Написание политик

- # Методы политики
- # Ответы политики
- # Методы политики без моделей
- # Гостевые пользователи
- # Фильтры политики

## # Авторизация действий с помощью политик

- # Авторизация действий с помощью политик через модель User
- # Авторизация действий с помощью политик через фасад gate
- # Авторизация действий с помощью политик через посредника
- # Авторизация действий с помощью политик через шаблоны Blade
- # Предоставление политикам дополнительного контекста

## # Авторизация и Inertia

## # Введение

Помимо встроенных служб [аутентификации](#), Laravel также предлагает простой способ авторизации действий пользователя с конкретными ресурсами. Например, даже если пользователь аутентифицирован, то он может быть не авторизован для

обновления или удаления определенных моделей Eloquent или записей базы данных вашего приложения. Функционал авторизации Laravel обеспечивает простой и организованный способ управления этими проверками авторизации.

Laravel предлагает два основных способа авторизации действий: [шлюзы](#) и [политики](#). Думайте о шлюзах и политиках, как о маршрутах и контроллерах. Шлюзы обеспечивают простой подход к авторизации, основанный на замыкании, в то время как политики, также как контроллеры, группируют логику вокруг конкретной модели или ресурса. В этой документации мы сначала рассмотрим шлюзы, а затем политики.

Вам не нужно выбирать между использованием исключительно Gates (шлюзов) или исключительно Policies (политик) при создании приложения. большинстве приложений, скорее всего, будет использоваться комбинация обоих, и это совершенно нормально! Gates наиболее подходят для действий, не связанных с какой-либо моделью или ресурсом, например, для просмотра панели управления администратора. В свою очередь, Policies следует использовать, когда вы хотите авторизовать действие для конкретной модели или ресурса.

## # Шлюзы (Gate)

### Написание шлюзов

Шлюзы – отличный способ изучить основы функционала авторизации Laravel; однако при создании надежных приложений Laravel, вам следует рассмотреть возможность использования [политик](#) для организации ваших правил авторизации.

Шлюз – это просто замыкание, которое определяет, имеет ли пользователь право выполнять указанное действие. Обычно шлюзы определяются в методе `boot` класса `App\Providers\AppServiceProvider` с использованием фасада `Gate`. Шлюзы всегда получают экземпляр пользователя в качестве своего первого аргумента и могут получать дополнительные аргументы, например, модель Eloquent.

В этом примере мы определим шлюз, решающий, может ли пользователь обновить указанную модель `App\Models\Post`. Шлюз выполнит это, сравнив идентификатор пользователя с идентификатором `user_id` пользователя, создавшего пост:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Запуск любых служб приложений.
 */
public function boot(): void
{
    $this->registerPolicies();

    Gate::define('update-post', function (User $user, Post $post) {
        return $user->id === $post->user_id;
    });
}
```

Шлюзы также могут быть определены с использованием callback-массива:

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Запуск любых служб приложений.
 */
public function boot(): void
{
    $this->registerPolicies();

    Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

## Авторизация действий через шлюзы

Чтобы авторизовать действие с помощью шлюзов, вы должны использовать методы `allows` или `denies` фасада `Gate`. Обратите внимание, что вам не требуется передавать в эти методы аутентифицированного в данный момент пользователя. Laravel автоматически позаботится о передаче пользователя в замыкание шлюза. Обычно методы авторизации шлюза вызываются в контроллерах вашего приложения перед выполнением действия, требующего авторизации:

```

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Обновить переданный пост.
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if (! Gate::allows('update-post', $post)) {
            abort(403);
        }

        // Обновление поста ...

        return redirect('/posts');
    }
}

```

Если вы хотите определить, авторизован ли другой (не аутентифицированный в настоящий момент) пользователь для выполнения действия, то вы можете использовать метод `forUser` фасада `Gate`:

```

if (Gate::forUser($user)->allows('update-post', $post)) {
    // Пользователь может обновить пост ...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // Пользователь не может обновить пост ...
}

```

Вы можете определить авторизацию нескольких действий одновременно, используя методы `any` или `none`:

```

if (Gate::any(['update-post', 'delete-post'], $post)) {
    // Пользователь может обновить или удалить пост ...
}

```

```
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // Пользователь не может обновить или удалить пост ...
}
```

## Авторизация или выброс исключений

Если вы хотите попытаться авторизовать действие и автоматически выдать исключение `Illuminate\Auth\Access\AuthorizationException`, если пользователю не разрешено выполнять данное действие, вы можете использовать метод `authorize` фасада `Gate`. Экземпляры `AuthorizationException` автоматически преобразуются Laravel в HTTP-ответ 403:

```
Gate::authorize('update-post', $post);

// Действие разрешено ...
```

## Предоставление дополнительного контекста шлюзом

Методы шлюза для авторизации полномочий (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) и [директивы авторизации Blade](#) (`@can`, `@cannot`, `@canany`) могут получать массив в качестве второго аргумента. Эти элементы массива передаются в качестве параметров замыканию шлюза и могут использоваться как дополнительный контекст при принятии решений об авторизации:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category, bool $pinned) {
    if (! $user->canPublishToGroup($category->group)) {
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false;
    }

    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
```

```
// Пользователь может создать пост ...
}
```

## Ответы шлюза

До сих пор мы рассматривали шлюзы, возвращающие простые логические значения. По желанию можно вернуть более подробный ответ, содержащий также сообщение об ошибке. Для этого вы можете вернуть экземпляр [Illuminate\Auth\Access\Response](#) из вашего шлюза:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::deny('Вы должны быть администратором.');
});
```

Даже когда вы возвращаете ответ авторизации из вашего шлюза, метод [Gate::allows](#) все равно будет возвращать простое логическое значение; однако вы можете использовать метод [Gate::inspect](#), чтобы получить полный возвращенный шлюзом ответ авторизации:

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // Действие разрешено ...
} else {
    echo $response->message();
}
```

При использовании метода [Gate::authorize](#), генерирующего исключение [AuthorizationException](#) для неавторизованного действия, сообщение об ошибке из ответа авторизации будет передано в HTTP-ответ:

```
Gate::authorize('edit-settings');
```

```
// Действие разрешено ...
```

## Настройка статуса HTTP-ответа

Когда доступ к действию запрещен через Gate, возвращается HTTP-ответ с кодом **403**. Однако иногда может быть полезно возвращать другой HTTP-статус. Вы можете настроить код статуса HTTP, который возвращается при неудачной проверке авторизации, используя статический конструктор `denyWithStatus` в классе `Illuminate\Auth\Access\Response`:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::denyWithStatus(404);
});
```

Поскольку скрытие ресурсов с помощью ответа **404** является общепринятым подходом в веб-приложениях, для удобства предлагается метод `denyAsNotFound`:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::denyAsNotFound();
});
```

## Хуки шлюзов

Иногда бывает необходимо предоставить все полномочия конкретному пользователю. Вы можете использовать метод `before` для определения замыкания, которое выполняется перед всеми другими проверками авторизации:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::before(function (User $user, string $ability) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

Если замыкание `before` возвращает результат, отличный от `null`, то этот результат и будет считаться результатом проверки авторизации.

Вы можете использовать метод `after` для определения замыкания, которое будет выполнено после всех других проверок авторизации:

```
use App\Models\User;

Gate::after(function (User $user, string $ability, bool|null $result, mixed $arguments) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

Значения, возвращаемые замыканиями `after`, не будут переопределять результат проверки авторизации, если шлюз или политика не возвратят `null`.

## Встроенная авторизация

Иногда вы можете захотеть определить, авторизован ли текущий аутентифицированный пользователь для выполнения данного действия без написания специального шлюза, соответствующего этому действию. Laravel позволяет вам выполнять эти типы «встроенных» проверок авторизации с помощью методов `Gate::allowIf` и `Gate::denyIf`. Встроенная авторизация не выполняет никаких определенных “before” или “after” хуков авторизации:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::allowIf(fn (User $user) => $user->isAdministrator());
```

```
Gate::denyIf(fn (User $user) => $user->banned());
```

Если действие не авторизовано или ни один пользователь в настоящее время не аутентифицирован, Laravel автоматически выдаст исключение [Illuminate\Auth\Access\AuthorizationException](#). Экземпляры [AuthorizationException](#) автоматически преобразуются в HTTP-ответ 403 обработчиком исключений Laravel:

## # Создание политик

### Генерация политик

Политики – это классы, которые организуют логику авторизации для конкретной модели или ресурса. Например, если ваше приложение является блогом, то у вас может быть модель [App\Models\Post](#) и соответствующая политика [App\Policies\PostPolicy](#) для авторизации действий пользователя, например, создание или обновление постов.

Чтобы сгенерировать новую политику, используйте команду [make:policy Artisan](#). Эта команда поместит новый класс политики в каталог [app/Policies](#) вашего приложения. Если этот каталог еще не существует, то Laravel предварительно создаст его:

```
php artisan make:policy PostPolicy
```

Команда [make:policy](#) сгенерирует пустой класс политики. Если вы хотите создать класс с заготовками методов политики, связанных с просмотром, созданием, обновлением и удалением ресурса, то вы можете указать параметр [--model](#) при выполнении команды:

```
php artisan make:policy PostPolicy --model=Post
```

### Регистрация политик

### Обнаружение политики

По умолчанию Laravel автоматически обнаруживает политики, если модель и политика соответствуют стандартным соглашениям об именах Laravel. В частности, политики должны находиться в каталоге `Policies`, расположенном в каталоге, содержащем ваши модели, или выше него. Так, например, модели могут быть размещены в каталоге `app/Models`, а политики — в каталоге `app/Policies`. В этой ситуации Laravel проверит наличие политик в `app/Models/Policies`, а затем в `app/Policies`. Кроме того, имя политики должно совпадать с названием модели и иметь суффикс `Policy`. Таким образом, модель `User` будет соответствовать классу политики `UserPolicy`.

Если вы хотите определить свою собственную логику обнаружения политики, вы можете зарегистрировать обратный вызов обнаружения собственной политики с помощью метода `Gate::guessPolicyNamesUsing`. Обычно этот метод следует вызывать из метода `boot AppServiceProvider` вашего приложения:

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function (string $modelClass) {
    // Return the name of the policy class for the given model...
});
```

## Регистрация политик вручную

Используя фасад `Gate`, вы можете вручную регистрировать политики и соответствующие им модели в методе `boot AppServiceProvider` вашего приложения:

```
use App\Models\Order;
use App\Policies\OrderPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Gate::policy(Order::class, OrderPolicy::class);
}
```

## # Написание политик

## Методы политики

После регистрации класса политики вы можете добавить методы для каждого из авторизуемых действий. Например, давайте определим метод `update` в нашем классе `PostPolicy`, который решает, может ли пользователь обновить указанный экземпляр поста.

Метод `update` получит в качестве аргументов экземпляры `User` и `Post` и должен вернуть `true` или `false`, которые будут указывать, авторизован ли пользователь обновлять указанный пост. Итак, в этом примере мы проверим, что идентификатор пользователя совпадает с `user_id` поста:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Определить, может ли пользователь обновить пост.
     */
    public function update(User $user, Post $post): bool
    {
        return $user->id === $post->user_id;
    }
}
```

Вы можете продолжить определение в политике необходимых методов дополнительных авторизуемых действий. Например, вы можете определить методы `view` или `delete` для авторизации различных действий, связанных с `Post`. Помните, что вы можете дать своим методам политики любые желаемые имена.

Если вы использовали опцию `--model` при создании своей политики через Artisan, то она уже будет содержать методы для следующих действий: `viewAny`, `view`, `create`, `update`, `delete`, `restore`, и `forceDelete`.

Все политики извлекаются через [контейнер служб Laravel](#), что позволяет вам объявлять любые

необходимые зависимости в конструкторе политики для их автоматического внедрения.

## Ответы политики

До сих пор мы рассматривали методы политики, возвращающие простые логические значения. По желанию можно вернуть более подробный ответ, содержащий также сообщение об ошибке. Для этого вы можете вернуть экземпляр `Illuminate\Auth\Access\Response` из вашего метода политики:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Определить, может ли пользователь обновить пост.
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
        ? Response::allow()
        : Response::deny('You do not own this post.');
}
```

При возврате ответа авторизации из вашей политики метод `Gate::allows` все равно будет возвращать простое логическое значение; однако вы можете использовать метод `Gate::inspect`, чтобы получить полный возвращенный шлюзом ответ авторизации:

```
use Illuminate\Support\Facades\Gate;

$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // Действие разрешено ...
} else {
    echo $response->message();
}
```

При использовании метода `Gate::authorize`, генерирующего исключение `AuthorizationException` для неавторизованного действия, сообщение об ошибке из ответа авторизации будет передано в HTTP-ответ:

```
Gate::authorize('update', $post);

// Действие разрешено ...
```

## Настройка статуса HTTP-ответа

Когда действие запрещается методом политики, возвращается ответ HTTP со статусом `403`. Однако иногда может быть полезно вернуть другой статус HTTP. Вы можете настроить код статуса HTTP, возвращаемый при неудачной проверке авторизации, используя статический конструктор `denyWithStatus` в классе `Illuminate\Auth\Access\Response`:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Определяет, может ли данный пользователь обновить указанный пост.
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
        ? Response::allow()
        : Response::denyWithStatus(404);
}
```

Поскольку скрытие ресурсов с помощью ответа `404` является общепринятым подходом в веб-приложениях, для удобства предлагается метод `denyAsNotFound`:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Определяет, может ли данный пользователь обновить указанный пост.
 */
public function update(User $user, Post $post): Response
{
```

```
        return $user->id === $post->user_id
            ? Response::allow()
            : Response::denyAsNotFound();
    }
}
```

## Методы политики без моделей

Некоторые методы политики получают только экземпляр аутентифицированного в данный момент пользователя. Эта ситуация наиболее распространена при авторизации действий `create`. Например, если вы создаете блог, то вы можете определить, имеет ли пользователь право вообще создавать какие-либо посты. В этих ситуациях ваш метод политики должен рассчитывать только на получение экземпляра пользователя:

```
/**
 * Определить, может ли пользователь создать пост.
 */
public function create(User $user): bool
{
    return $user->role == 'writer';
}
```

## Гостевые пользователи

По умолчанию все шлюзы и политики автоматически возвращают `false`, если входящий HTTP-запрос был инициирован не аутентифицированным пользователем. Однако вы можете разрешить прохождение этих проверок авторизации к вашим шлюзам и политикам, пометив в аргументе метода объявленный тип `User` как обнуляемый, путём добавления префикса в виде знака вопроса (?). Это означает, что значение может быть как объявленного типа `User`, так и быть равным `null`:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Определить, может ли пользователь обновить пост.
     *
```

```
 */
public function update(?User $user, Post $post): bool
{
    return $user?->id === $post->user_id;
}
}
```

## Фильтры политики

Для определенных пользователей вы можете разрешить все действия в рамках конкретной политики. Для этого определите в политике метод `before`. Метод `before` будет выполнен перед любыми другими методами в политике, что даст вам возможность авторизовать действие до фактического вызова предполагаемого метода политики. Этот функционал чаще всего используется для авторизации администраторов приложения на выполнение любых действий:

```
use App\Models\User;

/**
 * Выполнить предварительную авторизацию.
 */
public function before(User $user, string $ability): bool|null
{
    if ($user->isAdministrator()) {
        return true;
    }

    return null;
}
```

Если вы хотите отклонить все проверки авторизации для определенного типа пользователей, вы можете вернуть `false` из метода `before`. Если возвращается `null`, то проверка авторизации перейдет к методу политики.

Метод `before` класса политики не будет вызываться, если класс не содержит метода с именем, совпадающим с именем проверяемого полномочия.

# # Авторизация действий с помощью политик

## Авторизация действий с помощью политик через модель User

Модель `App\Models\User` приложения Laravel включает два полезных метода авторизации действий: `can` и `cannot`. Методы `can` и `cannot` получают имя действия, которое вы хотите авторизовать, и соответствующую модель. Например, давайте определим, авторизован ли пользователь для обновления переданной модели `App\Models\Post`. Обычно это делается в методе контроллера:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Обновить переданный пост.
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }

        // Обновление поста ...

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

Если для данной модели [политика зарегистрирована](#), то метод `can` автоматически вызовет соответствующую политику и вернет логический результат. Если для модели не зарегистрирована политика, то метод `can` попытается вызвать шлюз на основе замыкания, соответствующий переданному имени действия.

## Авторизация действий, не требующих моделей, с помощью политик через модель User

Помните, что некоторые действия могут соответствовать методам политики, например `create`, которые не требуют экземпляра модели. В этих ситуациях вы можете передать имя класса методу `can`. Имя класса будет использоваться для определения того, какую политику использовать при авторизации действия:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Сохранить пост.
     */
    public function store(Request $request): RedirectResponse
    {
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }

        // Сохранение поста ...

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

## Авторизация действий с помощью политик через через фасад gate

В дополнение к полезным методам, предоставляемым модели `App\Models\User`, вы всегда можете авторизовать действия с помощью метода `authorize` фасада `Gate`.

Подобно методу `can`, этот метод принимает имя действия, которое вы хотите авторизовать, и соответствующую модель. Если действие не авторизовано, то метод `authorize` выбросит исключение `Illuminate\Auth\Access\AuthorizationException`,

которое обработчик исключений Laravel автоматически преобразует в [403](#)

HTTP-ответ:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Обновить переданный пост.
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        Gate::authorize('update', $post);

        // Текущий пользователь может обновить пост в блоге ...

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

## Авторизация действий, не требующих моделей, с помощью политик через помощников контроллера

Как обсуждалось ранее, некоторые методы политики, например `create`, не требуют экземпляра модели. В таких ситуациях вы должны передать имя класса методу `authorize`. Имя класса будет использоваться для определения того, какую политику использовать при авторизации действия:

```
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

/**
```

```
* Создайте новый пост в блоге.  
*  
* @throws \Illuminate\Auth\Access\AuthorizationException  
*/  
public function create(Request $request): RedirectResponse  
{  
    Gate::authorize('create', Post::class);  
  
    // Текущий пользователь может создавать посты в блоге ...  
  
    return redirect('/posts');  
}
```

## Авторизация действий с помощью политик через посредника

Laravel содержит посредника, который может авторизовать действия до того, как входящий запрос достигнет ваших маршрутов или контроллеров. По умолчанию посреднику `Illuminate\Auth\Middleware\Authorize` может быть прикреплено к маршруту с помощью `can` [псевдонимом промежуточного программного обеспечения](#), который автоматически регистрируется в Laravel. Давайте рассмотрим пример использования посредника `can` для авторизации того, что пользователь может обновлять пост:

```
use App\Models\Post;  
  
Route::put('/post/{post}', function (Post $post) {  
    // Текущий пользователь может обновить пост ...  
})->middleware('can:update,post');
```

В этом примере мы передаем посреднику `can` два аргумента. Первый – это имя действия, которое мы хотим авторизовать, а второй – параметр маршрута, передаваемый методу политики. В этом случае поскольку мы используем [неявную привязку модели](#), то методу политики будет передана модель `App\Models\Post`. Если пользователь не авторизован для выполнения указанного действия, то посредник вернет ответ HTTP с кодом состояния `403`.

Для удобства вы также можете прикрепить посредник `can` к своему маршруту, используя метод `can`:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->can('update', 'post');
```

## Авторизация действий, не требующих моделей, с помощью политик через посредника

Опять же, некоторые методы политики, например `create`, не требуют экземпляра модели. В этих ситуациях вы можете передать имя класса посреднику. Имя класса будет использоваться для определения того, какую политику использовать при авторизации действия:

```
Route::post('/post', function () {
    // Текущий пользователь может создавать посты ...
})->middleware('can:create,App\Models\Post');
```

Указание полного имени класса в определении посредника строки может стать обременительным. По этой причине вы можете присоединить посредник `can` к вашему маршруту, используя метод `can`:

```
use App\Models\Post;

Route::post('/post', function () {
    // The current user may create posts...
})->can('create', Post::class);
```

## Авторизация действий с помощью политик через шаблоны Blade

При написании шаблонов Blade бывает необходимо отобразить часть страницы только в том случае, если пользователь авторизован для выполнения конкретного действия. Например, вы можете показать форму обновления поста в блоге, только если пользователь действительно уполномочен обновить сообщение. В этой ситуации вы можете использовать директивы `@can` и `@cannot`:

```
@can('update', $post)
    <!-- Текущий пользователь может обновить пост ... -->
@elsecan('create', App\Models\Post::class)
    <!-- Текущий пользователь может создавать новые посты ... -->
@endif
<!-- ... --&gt;
@endcan

@cannot('update', $post)
    &lt;!-- Текущий пользователь не может обновить пост ... --&gt;
@elsecannot('create', App\Models\Post::class)
    &lt;!-- Текущий пользователь не может создавать новые посты ... --&gt;
@endcannot</pre>
```

Эти директивы являются удобными ярлыками выражений [@if](#) и [@unless](#). Приведенные выше директивы [@can](#) и [@cannot](#) эквивалентны следующим выражениям:

```
@if (Auth::user()->can('update', $post))
    <!-- Текущий пользователь может обновить пост ... -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- Текущий пользователь не может обновить пост ... -->
@endunless
```

Вы также можете определить, авторизован ли пользователь для выполнения любого из указанных в массиве действий. Для этого используйте директиву [@canany](#):

```
@canany(['update', 'view', 'delete'], $post)
    <!-- Текущий пользователь может обновить, просмотреть или удалить пост ... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- Текущий пользователь может создать пост ... -->
@endcanany
```

**Авторизация действий, не требующих моделей, с помощью политик через шаблоны Blade**

Как и большинство других методов авторизации, вы можете передать имя класса в директивы `@can` и `@cannot`, если для действия не требуется экземпляр модели:

```
@can('create', App\Models\Post::class)
    <!-- Текущий пользователь может создавать посты ... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- Текущий пользователь не может создавать посты ... -->
@endcannot
```

## Предоставление политикам дополнительного контекста

При авторизации действий с использованием политик вы можете передать массив в качестве второго аргумента различным функциям авторизации и помощникам. Первый элемент в массиве будет использоваться для определения того, какая политика должна быть вызвана, в то время как остальные элементы массива передаются как параметры методу политики и могут использоваться как дополнительный контекст при принятии решений об авторизации. Например, рассмотрим `PostPolicy` и следующее определение метода, содержащего дополнительный параметр `$category`:

```
/**
 * Определить, может ли пользователь обновить пост.
 */
public function update(User $user, Post $post, int $category): bool
{
    return $user->id === $post->user_id &&
        $user->canUpdateCategory($category);
}
```

При попытке определить, может ли аутентифицированный пользователь обновить указанный пост, мы можем вызвать этот метод политики следующим образом:

```
/**
 * Обновить конкретный пост.
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post): RedirectResponse
```

```
{  
    Gate::authorize('update', [$post, $request->category]);  
  
    // Текущий пользователь может обновить пост в блоге ...  
  
    return redirect('/posts');  
}
```

## # Авторизация и Inertia

Хотя авторизация всегда должна обрабатываться на сервере, часто бывает удобно предоставить вашему внешнему приложению данные авторизации, чтобы правильно отобразить пользовательский интерфейс вашего приложения. Laravel не определяет необходимое соглашение для предоставления информации об авторизации интерфейсу на базе Inertia.

However, if you are using one of Laravel's Inertia-based [starter kits](#), your application already contains a [HandleInertiaRequests](#) middleware. Within this middleware's `share` method, you may return shared data that will be provided to all Inertia pages in your application. This shared data can serve as a convenient location to define authorization information for the user: Однако, если вы используете один из [стартовых наборов](#) Laravel на основе Inertia, ваше приложение уже содержит посредника [HandleInertiaRequests](#). В рамках метода `share` этого посредника вы можете возвращать общие данные, которые будут предоставлены всем страницам Inertia в вашем приложении. Эти общие данные могут служить удобным местом для определения информации об авторизации пользователя:

```
<?php  
  
namespace App\Http\Middleware;  
  
use App\Models\Post;  
use Illuminate\Http\Request;  
use Inertia\Middleware;  
  
class HandleInertiaRequests extends Middleware  
{  
    // ...  
  
    /**  
     * Define the props that are shared by default.  
     *  
     * @return array<string, mixed>
```

```
 */
public function share(Request $request)
{
    return [
        ...parent::share($request),
        'auth' => [
            'user' => $request->user(),
            'permissions' => [
                'post' => [
                    'create' => $request->user()->can('create', Post::class),
                ],
            ],
        ],
    ];
}
}
```

# Подтверждение адреса электронной почты

## # Введение

- # Подготовка модели
- # Подготовка базы данных

## # Маршрутизация

- # Уведомление о подтверждении электронной почты
- # Обработчик проверки электронной почты
- # Повторная отправка письма с подтверждением
- # Защита маршрутов

## # Настройка

## # События

## # Введение

Многие веб-приложения требуют от пользователей подтверждения своего адреса электронной почты перед использованием приложения. Вместо того чтобы заставлять вас самостоятельно реализовывать этот функционал повторно для каждого создаваемого вами приложения, Laravel предлагает удобные встроенные службы для отправки и проверки запросов подтверждения адреса электронной почты.

Хотите быстро начать? Установите один из [стартовых комплектов](#) в новое приложение Laravel. Стартовые комплекты позаботятся о построении всей вашей системы аутентификации, включая поддержку подтверждения электронной почты.

## Подготовка модели

Убедитесь, что ваша модель `App\Models\User` реализует контракт `Illuminate\Contracts\Auth\MustVerifyEmail`:

```
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

Как только этот интерфейс будет добавлен в вашу модель, вновь зарегистрированным пользователям будет автоматически отправлено электронное письмо со ссылкой для подтверждения адреса электронной почты. Это происходит легко, потому что Laravel автоматически регистрирует `Illuminate\Auth\Listeners\SendEmailVerificationNotification` слушатель для события `Illuminate\Auth\Events\Registered`.

Если вы самостоятельно выполняете регистрацию в своем приложении вместо использования стартового комплекта, то вы должны убедиться, что запускаете событие `Illuminate\Auth\Events\Registered` после успешной регистрации пользователя:

```
use Illuminate\Auth\Events\Registered;

event(new Registered($user));
```

## Подготовка базы данных

Ваша таблица `users` должна содержать столбец `email_verified_at` для сохранения даты и времени подтверждения адреса электронной почты пользователем. Обычно это включено в миграцию базы данных Laravel `0001_01_01_000000_create_users_table.php` по умолчанию.

## # Маршрутизация

Чтобы правильно реализовать подтверждение электронной почты, необходимо определить три маршрута. Во-первых, потребуется маршрут для отображения уведомления пользователю о том, что он должен щелкнуть ссылку подтверждения электронной почты в письме, которое Laravel отправит ему после регистрации.

Во-вторых, потребуется маршрут для обработки запросов, сгенерированных, когда пользователь щелкает ссылку подтверждения электронной почты в электронном письме.

В-третьих, потребуется маршрут для повторной отправки ссылки для подтверждения, если пользователь случайно потеряет первую ссылку для подтверждения.

## Уведомление о подтверждении электронной почты

Как упоминалось ранее, должен быть определен маршрут, возвращающий страницу, инструктирующую пользователя щелкнуть ссылку для подтверждения электронной почты, которая была отправлена ему Laravel по электронной почте после регистрации. Эта страница будет отображаться для пользователей, когда они попытаются получить доступ к другим частям приложения без предварительной проверки своего адреса электронной почты. Помните, что ссылка автоматически отправляется пользователю по электронной почте, если ваша модель `App\Models\User` реализует интерфейс `MustVerifyEmail`:

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

Маршрут, который возвращает уведомление о подтверждении по электронной почте, должен называться `verification.notice`. Важно, чтобы маршруту было присвоено это точное имя, поскольку посредник `verify`, [включенный в Laravel](#), будет автоматически перенаправлять на это имя маршрута, если пользователь не подтвердил свой адрес электронной почты.

При выполнении проверки электронной почты самостоятельно, вам необходимо определить

содержание страницы уведомления о проверке. Если вам необходим каркас, включающий все необходимые страницы для аутентификации и проверки, ознакомьтесь со [стартовыми комплектами приложений Laravel](#).

## Обработчик проверки электронной почты

Затем, нам нужно определить маршрут, обрабатывающий запросы, сгенерированные, когда пользователь щелкает ссылку подтверждения электронной почты, которая была отправлена ему по электронной почте. Этот маршрут должен называться `verification.verify` и ему должны быть назначены посредники `auth` и `signed`:

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;

Route::get('/email/verify/{id}/{hash}', function (EmailVerificationRequest $request)
    $request->fulfill();

    return redirect('/');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

Прежде чем двигаться дальше, давайте подробнее рассмотрим этот маршрут. Во-первых, вы заметите, что мы используем тип запроса `EmailVerificationRequest` вместо типичного экземпляра `Illuminate\Http\Request`. `EmailVerificationRequest` – это [запрос формы](#), который включен в Laravel. Этот запрос автоматически позаботится о проверке параметров запроса `id` и `hash`.

Далее, мы можем приступить непосредственно к вызову метода `fulfill` запроса. Этот метод вызовет метод `markEmailAsVerified` для аутентифицированного пользователя и запустит событие `Illuminate\Auth\Events\Verified`. Метод `markEmailAsVerified` доступен для модели по умолчанию `App\Models\User` через базовый класс `Illuminate\Foundation\Auth\User`. После подтверждения адреса электронной почты пользователя вы можете перенаправить его куда пожелаете.

## Повторная отправка письма с подтверждением

Иногда пользователь может потерять или случайно удалить письмо с подтверждением адреса электронной почты. Чтобы учесть это, вы можете определить маршрут, позволяющий пользователю запрашивать повторную отправку письма с подтверждением. Затем, вы можете сделать запрос по этому маршруту, поместив простую кнопку отправки формы на [странице уведомления о подтверждении](#):

```
use Illuminate\Http\Request;

Route::post('/email/verification-notification', function (Request $request) {
    $request->user()->sendEmailVerificationNotification();

    return back()->with('message', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
```

## Защита маршрутов

[Middleware](#) может использоваться для разрешения доступа только для проверенных пользователей к определенному маршруту. Laravel включает [verified псевдоним middleware](#), который является псевдонимом для класса middleware `Illuminate\Auth\Middleware\EnsureEmailIsVerified`. Поскольку этот псевдоним уже автоматически зарегистрирован Laravel, все, что вам нужно сделать, это прикрепить `verified` middleware к определению маршрута. Обычно этот middleware используется вместе с middleware `auth`:

```
Route::get('/profile', function () {
    // Только подтвержденные пользователи могут получить доступ к этому маршруту ...
})->middleware(['auth', 'verified']);
```

Если непроверенный пользователь попытается получить доступ к маршруту, которому назначен этот посредник, то он будет автоматически перенаправлен на [именованный маршрут](#) `verification.notice`.

## # Настройка

### Настройка подтверждения адреса электронной почты

Хотя уведомление о подтверждении электронной почты по умолчанию должно удовлетворять требованиям большинства приложений, Laravel позволяет вам изменить сообщение подтверждения электронной почты.

Для начала, передайте замыкание методу `toMailUsing` уведомления `Illuminate\Auth\Notifications\VerifyEmail`. Замыкание получит экземпляр модели, содержащий уведомление, а также подписанный URL-адрес подтверждения электронной почты, который пользователь должен посетить для проверки адреса электронной почты. Замыкание должно вернуть экземпляр `Illuminate\Notifications\Messages\MailMessage`. Как правило, вызов метода `toMailUsing` осуществляется в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    // ...

    VerifyEmail::toMailUsing(function (object $notifiable, string $url) {
        return (new MailMessage)
            ->subject('Verify Email Address')
            ->line('Click the button below to verify your email address.')
            ->action('Verify Email Address', $url);
    });
}
```

Чтобы узнать больше о почтовых уведомлениях, обратитесь к [документации по почтовым уведомлениям](#).

## # События

При использовании [стартовых комплектов](#) Laravel запускает `Illuminate\Auth\Events\Verified` событие в процессе проверки электронной почты.

Если вы самостоятельно обрабатываете проверку электронной почты для своего приложения, то вы должны запускать эти события после завершения проверки.

# Шифрование

- # Введение
- # Конфигурирование
  - # Плавная смена ключей шифрования
- # Использование шифровальщика

## # Введение

Сервисы шифрования Laravel предоставляют простой и удобный интерфейс для шифрования и дешифрования текста через OpenSSL с использованием шифрования AES-256 и AES-128. Все зашифрованные значения Laravel подписываются с использованием кода аутентификации сообщения (MAC), поэтому их базовое значение не может быть изменено или подделано после шифрования.

## # Конфигурирование

Перед использованием шифровальщика Laravel вы должны установить параметр `key` в конфигурационном файле `config/app.php`. Это значение конфигурации управляет переменной окружения `APP_KEY`. Вы должны использовать команду `php artisan key:generate` для генерации значения этой переменной, поскольку команда `key:generate` будет использовать безопасный генератор случайных байтов PHP для создания криптографически безопасного ключа для вашего приложения. Обычно значение переменной среды `APP_KEY` генерируется для вас во время [установки Laravel](#).

## Плавная смена ключей шифрования

При изменении ключа шифрования вашего приложения все сеансы аутентификации пользователей будут завершены. Это происходит потому, что каждый cookie, включая сессионные, шифруется Laravel. Кроме того, данные, зашифрованные с использованием предыдущего ключа, больше не будут доступны для расшифровки.

Для решения этой проблемы Laravel предоставляет возможность указать предыдущие ключи шифрования в переменной окружения `APP_PREVIOUS_KEYS` вашего приложения. Эта переменная может содержать список предыдущих ключей, разделенных запятыми:

```
APP_KEY="base64:J63qRTDLub5NuZvP+kb8YIorGS6qFYHKVo6u7179stY="
APP_PREVIOUS_KEYS="base64:2nLsGFGzyoae2ax3EF2Lyq/hH6QghBGLIq5uL+Gp8/w="
```

Когда вы устанавливаете эту переменную окружения, Laravel всегда будет использовать "текущий" ключ для шифрования данных. Однако при расшифровке Laravel сначала попытается использовать текущий ключ, а если это не удастся, то попробует все предыдущие ключи, пока не найдет подходящий.

Этот метод плавной расшифровки позволяет пользователям продолжать использовать ваше приложение без перерывов, даже если вы поменяли ключ шифрования.

## # Использование шифровальщика

### Шифрование значения

Вы можете зашифровать значение, используя метод `encryptString` фасада `Crypt`. Все значения будут зашифрованы с использованием OpenSSL и шифра `AES-256-CBC`. Кроме того, все зашифрованные значения подписываются кодом аутентификации сообщения (MAC). Встроенный код аутентификации сообщений предотвратит расшифровку любых значений, которые были подделаны злоумышленниками:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class DigitalOceanTokenController extends Controller
{
    /**
     * Сохраните DigitalOcean API-токен пользователя.
     */
}
```

```
public function store(Request $request): RedirectResponse
{
    $request->user()->fill([
        'token' => Crypt::encryptString($request->token),
    ])->save();

    return redirect('/secrets');
}

}
```

## Расшифровка значения

Вы можете расшифровать значения, используя метод `decryptString` фасада `Crypt`. Если значение не может быть правильно расшифровано, например, когда код аутентификации сообщения недействителен, будет выброшено исключение `Illuminate\Contracts\Encryption\DecryptException`:

```
use Illuminate\Contracts\Encryption\DecryptException;
use Illuminate\Support\Facades\Crypt;

try {
    $decrypted = Crypt::decryptString($encryptedValue);
} catch (DecryptException $e) {
    // ...
}
```

# Хеширование

# Введение

# Конфигурирование

# Основы использования

# Хеширование паролей

# Проверка совпадения пароля с хешем

# Определение необходимости повторного хеширования пароля

# Проверка алгоритма хеширования

## # Введение

Фасад [Hash](#) фреймворка Laravel обеспечивает безопасное хеширование Bcrypt и Argon2 для хранения паролей пользователей. Если вы используете каркас одного из [стартовых комплектов приложений Laravel](#), то для регистрации и аутентификации по умолчанию будет использоваться Bcrypt.

Bcrypt – отличный выбор для хеширования паролей, потому что его «коэффициент работы» регулируется, а это означает, что время, необходимое для генерации хеш-кода, может быть увеличено по мере увеличения мощности оборудования. При хешировании паролей – чем медленнее, тем лучше. Чем больше времени требуется алгоритму для хеширования пароля, тем больше времени требуется злоумышленникам для создания «радужных таблиц» всех возможных строковых хеш-значений, которые могут использоваться в атаках.

## # Конфигурирование

По умолчанию Laravel использует драйвер [bcrypt](#) при хешировании данных. Однако поддерживаются несколько других драйверов хеширования, в том числе [argon](#) и [argon2id](#).

Вы можете указать драйвер хеширования вашего приложения, используя переменную среды [HASH\\_DRIVER](#). Но если вы хотите настроить все параметры

драйвера хеширования Laravel, вам следует опубликовать полный файл конфигурации хеширования с помощью Artisan-команды `config:publish`:

```
php artisan config:publish hashing
```

## # Основы использования

### Хеширование паролей

Вы можете хешировать пароль, вызвав метод `make` фасада `Hash`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class PasswordController extends Controller
{
    /**
     * Обновить пароль пользователя.
     */
    public function update(Request $request): RedirectResponse
    {
        // Проверить длину нового пароля ...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();

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

### Регулировка коэффициента работы Bcrypt

Если вы используете алгоритм Bcrypt, метод `make` позволяет вам управлять коэффициентом работы алгоритма с помощью параметра `rounds`; однако значение по умолчанию приемлемо для большинства приложений:

```
$hashed = Hash::make('password', [  
    'rounds' => 12,  
]);
```

## Регулировка коэффициента работы Argon2

Если вы используете алгоритм Argon2, метод `make` позволяет вам управлять коэффициентом работы алгоритма с помощью параметров `memory`, `time` и `threads`; однако значения по умолчанию приемлемы для большинства приложений:

```
$hashed = Hash::make('password', [  
    'memory' => 1024,  
    'time' => 2,  
    'threads' => 2,  
]);
```

Дополнительную информацию об этих параметрах можно найти в [официальной документации PHP](#).

## Проверка совпадения пароля с хешем

Метод `check` фасада `Hash` позволяет проверить, что указанная текстовая строка соответствует заданному хешу:

```
if (Hash::check('plain-text', $hashedPassword)) {  
    // Пароли совпадают ...  
}
```

## Определение необходимости повторного хеширования пароля

Метод `needsRehash` фасада `Hash` позволяет определить, изменился ли коэффициентом работы, используемый хешером, с момента хеширования пароля. Некоторые приложения предпочитают выполнять эту проверку во время процесса аутентификации приложения:

```
if (Hash::needsRehash($hashed)) {  
    $hashed = Hash::make('plain-text');  
}
```

## # Проверка алгоритма хеширования

Чтобы предотвратить манипуляции с алгоритмом хеширования, метод Laravel `Hash::check` сначала проверяет, что данный хэш был сгенерирован с использованием выбранного алгоритма хеширования приложения. Если алгоритмы разные, будет выброшено исключение `RuntimeException`.

Это ожидаемое поведение для большинства приложений, где не ожидается изменения алгоритма хеширования, а разные алгоритмы могут указывать на злонамеренную атаку. Однако если вам необходимо поддерживать несколько алгоритмов хеширования в вашем приложении, например, при переходе от одного алгоритма к другому, вы можете отключить проверку алгоритма хеширования, установив для переменной среды `HASH_VERIFY` значение `false`:

```
HASH_VERIFY=false
```

# Сброс пароля

## # Введение

- # Подготовка модели
- # Подготовка базы данных
- # Конфигурирование доверенных хостов

## # Маршрутизация

- # Запрос ссылки для сброса пароля
- # Сброс пароля

## # Удаление просроченных токенов

## # Настройка

## # Введение

Большинство веб-приложений предоставляют пользователям возможность сбросить забытые пароли. Вместо того чтобы заставлять вас заново реализовывать этот функционал самостоятельно для каждого создаваемого вами приложения, Laravel предлагает удобные сервисы для отправки ссылок для сброса пароля и, собственно, безопасного сброса паролей.

Хотите быстро начать? Установите один из [стартовых комплектов](#) в новое приложение Laravel. Стартовые комплекты позаботятся о построении всей вашей системы аутентификации, включая сброс забытых паролей.

## Подготовка модели

Перед использованием функционала сброса пароля Laravel модель вашего приложения `App\Models\User` должна использовать трейт

[Illuminate\Notifications\Notifiable](#). Обычно этот трейт уже содержится по умолчанию в модели [App\Models\User](#) при создании новых приложений Laravel.

Затем убедитесь, что ваша модель [App\Models\User](#) реализует контракт [Illuminate\Contracts\Auth\CanResetPassword](#). Модель [App\Models\User](#) Laravel, уже реализует этот интерфейс и использует трейт [Illuminate\Auth\Passwords\CanResetPassword](#), включающий методы, необходимые для реализации интерфейса.

## Подготовка базы данных

Необходимо создать таблицу для сохранения токенов сброса пароля вашего приложения. Обычно это включено в миграцию базы данных Laravel по умолчанию [0001\\_01\\_01\\_000000\\_create\\_users\\_table.php](#).

## Конфигурирование доверенных хостов

По умолчанию Laravel будет отвечать на все запросы, которые он получает, независимо от содержимого заголовка [Host](#) HTTP-запроса. Кроме того, значение заголовка [Host](#) будет использоваться при генерации абсолютных URL-адресов вашего приложения во время веб-запроса.

Как правило, вам следует настроить свой веб-сервер (Nginx или Apache), так, чтобы он обслуживал запросы, соответствующие только указанному имени хоста. Однако, если у вас нет возможности напрямую настроить свой веб-сервер и вам нужно указать Laravel, чтобы он отвечал только на определенные имена хостов, вы можете сделать это, используя метод промежуточного программного обеспечения [trustHosts](#) в файле [bootstrap/app.php](#) вашего приложения. Это особенно важно, когда ваше приложение предлагает функционал сброса пароля.

Чтобы узнать больше об этом методе посредника, обратитесь к [документации посредника TrustHosts](#).

## # Маршрутизация

Чтобы правильно реализовать поддержку, позволяющую пользователям сбрасывать свои пароли, нам нужно будет определить несколько маршрутов. Во-первых, нам понадобится пара маршрутов для обработки, позволяющей пользователю запрашивать ссылку для сброса пароля через свой адрес

электронной почты. Во-вторых, нам понадобится пара маршрутов для обработки фактического сброса пароля при посещении пользователем ссылки для сброса пароля, отправленной ему по электронной почте, и последующего заполнения формы сброса пароля.

## Запрос ссылки для сброса пароля

### Форма запроса ссылки для сброса пароля

Сначала мы определим маршруты, которые необходимы для запроса ссылок для сброса пароля. Для начала мы определим маршрут, который возвращает шаблон с формой запроса ссылки для сброса пароля:

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

Шаблон, возвращаемый этим маршрутом, должен иметь форму с полем `email` для указания адреса электронной почты, позволяющем пользователю запросить ссылку для сброса пароля.

### Обработка отправки формы

Затем мы определим маршрут, который обрабатывает запрос на отправку формы из шаблона `forgot-password`. Этот маршрут будет отвечать за проверку адреса электронной почты и отправку запроса на сброс пароля соответствующему пользователю:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

Route::post('/forgot-password', function (Request $request) {
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
        ? back()->with(['status' => __($status)])
        : back();
});
```

```
: back()->withErrors(['email' => __('$status')]);  
})->middleware('guest')->name('password.email');
```

Прежде чем двигаться дальше, давайте рассмотрим этот маршрут более подробно. Сначала проверяется атрибут запроса `email`. Затем мы будем использовать встроенный в Laravel «брокер паролей» через фасад `Password`, чтобы отправить пользователю ссылку для сброса пароля. Брокер паролей позаботится о получении пользователя по указанному полю (в данном случае по адресу электронной почты) и отправит пользователю ссылку для сброса пароля через встроенную [систему уведомлений](#) Laravel.

Метод `sendResetLink` возвращает ключ «status». Этот статус может быть переведен с помощью помощников [локализации](#) Laravel, чтобы показать пользователю удобное сообщение о статусе его запроса. Перевод статуса сброса пароля определяется языковым файлом `lang/{lang}/passwords.php` вашего приложения. Запись для каждого возможного значения ключа статуса находится в языковом файле `passwords`.

По умолчанию скелет приложения Laravel не включает каталог `lang`. Если вы хотите настроить языковые файлы Laravel, вы можете опубликовать их с помощью команды `lang:publish` Artisan.

Вам может быть интересно: как Laravel знает о том, как получить запись пользователя из базы данных вашего приложения при вызове метода `sendResetLink` фасада `Password`? Брокер паролей Laravel использует «поставщиков пользователей» вашей системы аутентификации для получения записей из базы данных. Поставщик пользователей, используемый брокером паролей, настраивается в массиве `passwords` вашего файла конфигурации `config/auth.php`. Чтобы узнать больше о создании пользовательских поставщиков служб, обратитесь к [документации по аутентификации](#).

При выполнении сброса пароля самостоятельно, вы должны сами определять содержимое страницы и маршрутов. Если вам необходим каркас,

включающий всю необходимую логику аутентификации и проверки, ознакомьтесь со [стартовыми комплектами приложений Laravel](#).

## Сброс пароля

### Форма сброса пароля

Затем мы определим маршруты, необходимые для фактического сброса пароля, когда пользователь щелкает ссылку для сброса пароля, отправленную ему по электронной почте, и предоставляет новый пароль. Во-первых, давайте определим маршрут, который будет отображать форму сброса пароля, после того как пользователь щелкает ссылку сброса пароля. Этот маршрут получит параметр `token`, который мы будем использовать позже для проверки запроса на сброс пароля:

```
Route::get('/reset-password/{token}', function (string $token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

Экран, возвращаемый этим маршрутом, должен отображать форму, содержащую поле `email`, поле `password`, поле `password_confirmation` и скрытое поле `token`, которое должно содержать значение секретного `$token`, полученного нашим маршрутом.

### Обработка отправки формы

Конечно, нам нужно определить маршрут для фактической обработки отправки формы сброса пароля. Этот маршрут будет отвечать за проверку входящего запроса и обновление пароля пользователя в базе данных:

```
use Illuminate\Auth\Events\PasswordReset;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;

Route::post('/reset-password', function (Request $request) {
    $request->validate([
        'token' => 'required',
```

```

'email' => 'required|email',
'password' => 'required|min:8|confirmed',
]);

$status = Password::reset(
    $request->only('email', 'password', 'password_confirmation', 'token'),
    function (User $user, string $password) {
        $user->forceFill([
            'password' => Hash::make($password)
        ])->setRememberToken(Str::random(60));

        $user->save();

        event(new PasswordReset($user));
    }
);

return $status === Password::PASSWORD_RESET
    ? redirect()->route('login')->with('status', __($status))
    : back()->withErrors(['email' => [__([$status])]]);
})->middleware('guest')->name('password.update');

```

Прежде чем двигаться дальше, давайте рассмотрим этот маршрут более подробно. Сначала проверяются атрибуты запроса `token`, `email`, и `password`. Далее мы будем использовать встроенный в Laravel «брокер паролей» (через фасад `Password`) для проверки учетных данных запроса сброса пароля.

Если токен, адрес электронной почты и пароль, переданные брокеру паролей, действительны, будет вызвано замыкание, переданное методу `reset`. В рамках этого замыкания, которое получает экземпляр пользователя и пароль в виде обычного текста из формы сброса пароля, мы можем обновить пароль пользователя в базе данных.

Метод `reset` возвращает ключ «`status`». Этот статус может быть переведен с помощью помощников [локализации](#) Laravel, чтобы показать пользователю удобное сообщение о статусе его запроса. Перевод статуса сброса пароля определяется языковым файлом `lang/{lang}/passwords.php` вашего приложения. Запись для каждого возможного значения ключа статуса находится в языковом файле `passwords`. Если ваше приложение не содержит каталога `lang`, вы можете создать его, используя команду Artisan `lang:publish`.

Прежде чем двигаться дальше, вам может быть интересно, как Laravel знает, как получить запись пользователя из базы данных вашего приложения при вызове

метода `reset` фасада `Password`. Брокер паролей Laravel использует «поставщиков пользователей» вашей системы аутентификации для получения записей из базы данных. Поставщик пользователей, используемый брокером паролей, настраивается в массиве `passwords` вашего файла конфигурации `config/auth.php`. Чтобы узнать больше о создании пользовательских поставщиков служб, обратитесь к [документации по аутентификации](#).

## # Удаление просроченных токенов

Токены сброса пароля с истекшим сроком действия будут по-прежнему присутствовать в вашей базе данных. Однако вы можете легко удалить эти записи, используя Artisan-команду `auth:clear-resets`:

```
php artisan auth:clear-resets
```

Если вы хотите автоматизировать этот процесс, рассмотрите возможность добавления команды в [планировщик](#) вашего приложения:

```
use Illuminate\Support\Facades\Schedule;  
  
Schedule::command('auth:clear-resets')->everyFifteenMinutes();
```

## # Настройка

### Настройка ссылки для сброса

Вы можете изменить URL-адрес ссылки для сброса пароля, используя метод `createUrlUsing` класса уведомлений `ResetPassword`. Этот метод принимает замыкание, которое получает экземпляр ожидающего уведомление пользователя, а также токен ссылки для сброса пароля. Как правило, вызов этого метода осуществляется в методе `boot` вашего поставщика служб `App\Providers\AppServiceProvider`:

```
use App\Models\User;  
use Illuminate\Auth\Notifications\ResetPassword;  
  
/**  
 * Запуск любых служб приложения.  
 */
```

```
 */
public function boot(): void
{
    $this->registerPolicies();

    ResetPassword::createUrlUsing(function (User $user, string $token) {
        return 'https://example.com/reset-password?token=' . $token;
    });
}
```

## Настройка уведомлений о сбросе пароля

Вы можете легко изменить класс уведомления, используемый для отправки пользователю ссылки для сброса пароля. Для начала переопределите метод `sendPasswordResetNotification` в модели `App\Models\User`. В этом методе вы можете отправить уведомление, используя любой [класс уведомлений](#), созданный вами. Токен для сброса пароля – это первый аргумент, получаемый методом. Вы можете использовать этот `$token` для создания URL сброса пароля по вашему усмотрению и для дальнейшей отправки уведомления пользователю:

```
use App\Notifications\ResetPasswordNotification;

/**
 * Отправить пользователю уведомление о сбросе пароля.
 *
 * @param string $token
 */
public function sendPasswordResetNotification($token): void
{
    $url = 'https://example.com/reset-password?token=' . $token;

    $this->notify(new ResetPasswordNotification($url));
}
```

# База данных · Начало работы

## # Введение

- # Конфигурирование
- # Соединения для чтения и записи

## # Выполнение SQL-запросов

- # Использование нескольких соединений к базе данных
- # Прослушивание событий запроса
- # Мониторинг общего времени выполнения запроса

## # Транзакции базы данных

- # Подключение к базе данных с помощью интерфейса командной строки Artisan
- # Инспектирование базы данных
- # Мониторинг баз данных

## # Введение

Почти каждое современное веб-приложение взаимодействует с базой данных. Laravel делает взаимодействие с базами данных чрезвычайно простым через поддержку множества баз данных, используя либо сырой SQL [построителя запросов](#), либо [Eloquent ORM](#). В настоящее время Laravel обеспечивает поддержку пяти баз данных:

- MariaDB 10.3+ ([Version Policy](#))
- MySQL 5.7+ ([Version Policy](#))
- PostgreSQL 10.0+ ([Version Policy](#))
- SQLite 3.26.0+
- SQL Server 2017+ ([Version Policy](#))

## Конфигурирование

Конфигурация служб баз данных Laravel находится в конфигурационном файле `config/database.php` вашего приложения. В этом файле вы можете определить все соединения к базе данных, а также указать, какое соединение должно использоваться по умолчанию. Большинство параметров конфигурации в этом файле определяется значениями переменных окружения вашего приложения. В этом файле представлены примеры для большинства систем баз данных, поддерживаемых Laravel.

По умолчанию пример [конфигурации окружения](#) Laravel готов к использованию с [Laravel Sail](#), который представляет собой конфигурацию Docker для разработки приложений Laravel на вашем локальном компьютере. Однако вы можете изменить конфигурацию своей базы данных по мере необходимости для своей локальной базы данных.

## Конфигурация SQLite

Базы данных SQLite содержатся в одном файле вашей файловой системы. Вы можете создать новую базу данных SQLite, используя команду `touch` в консоли: `touch database/database.sqlite`. После создания базы данных вы можете легко настроить переменные окружения так, чтобы они указывали на эту базу данных, указав абсолютный путь к базе данных в переменной `DB_DATABASE` окружения:

```
DB_CONNECTION=sqlite  
DB_DATABASE=/absolute/path/to/database.sqlite
```

По умолчанию ограничения внешнего ключа включены для соединений SQLite. Если вы хотите отключить их, вам следует установить для переменной среды `DB_FOREIGN_KEYS` значение `false`:

```
DB_FOREIGN_KEYS=false
```

Если вы используете [установщик Laravel](#) для создания приложения Laravel и выбираете SQLite в качестве базы данных, Laravel автоматически создаст `database/database.sqlite` и запустит для вас стандартную [миграцию базы данных](#).

# Конфигурация Microsoft SQL Server

Чтобы использовать базу данных Microsoft SQL Server, вы должны убедиться, что у вас установлены расширения PHP `sqlsrv` и `pdo_sqlsrv`, а также любые зависимости, которые могут им потребоваться, например, драйвер Microsoft SQL ODBC.

## Конфигурация с использованием URL

Обычно соединения с базой данных конфигурируются с использованием нескольких значений, таких как `host`, `database`, `username`, `password` и т.д. Каждое из этих значений имеет свою собственную соответствующую переменную окружения. Это означает, что при указании информации о соединении с базой данных на рабочем веб-сервере вам необходимо управлять несколькими переменными окружения.

Некоторые поставщики СУБД, такие, как AWS и Heroku, предоставляют единый «URL» базы данных, который содержит всю информацию о соединении в одной строке. Пример URL-адреса базы данных может выглядеть так:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

Эти URL обычно следуют соглашению стандартной схемы:

```
driver://username:password@host:port/database?options
```

Для удобства Laravel поддерживает эти URL-адреса в качестве альтернативы настройке базы данных с несколькими параметрами конфигурации. Если присутствует параметр конфигурации `url` (или соответствующая переменная `DB_URL` окружения), то он будет использоваться для получения информации о соединении с базой данных и об учетных данных.

## Соединения для чтения и записи

По желанию можно использовать одно соединение с базой данных для операторов `SELECT`, а другое – для операторов `INSERT`, `UPDATE` и `DELETE`. Laravel упрощает эту задачу, и всегда будут использоваться соответствующие соединения, независимо от того, используете ли вы сырье запросы построителя запросов или Eloquent ORM.

Чтобы увидеть, как должны быть настроены соединения для чтения / записи, давайте посмотрим на этот пример:

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,

    'database' => env('DB_DATABASE', 'laravel'),
    'username' => env('DB_USERNAME', 'root'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => env('DB_CHARSET', 'utf8mb4'),
    'collation' => env('DB_COLLATION', 'utf8mb4_unicode_ci'),
    'prefix' => '',
    'prefix_indexes' => true,
    'strict' => true,
    'engine' => null,
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
    ]) : [],
],
```

Обратите внимание, что в массив конфигурации были добавлены три ключа: `read`, `write` и `sticky`. Ключи `read` и `write` имеют значения массива, содержащие один ключ: `host`. Остальные параметры базы данных для соединений `read` и `write` будут объединены из основного массива конфигурации `mysql`.

В массивы `read` и `write` вам нужно помещать только те элементы, значения которых вы хотите переопределить из основного массива `mysql`. Таким образом, в этом случае `192.168.1.1` будет использоваться в качестве хоста для соединения «чтение», а `192.168.1.3` – для соединения «запись». Учетные данные БД, префикс, набор символов и все другие параметры из основного массива `mysql` будут совместно использоваться обоими соединениями. Если в массиве конфигурации

`host` существует несколько значений, то для каждого запроса хост базы данных будет выбран случайным образом.

## Параметр `sticky`

Параметр `sticky` – это *необязательное* значение, которое может использоваться для разрешения немедленного чтения записей, которые были записаны в базу данных во время текущего цикла запроса. Если опция `sticky` включена и в текущем цикле запроса к базе данных была выполнена операция «записи», то любые дальнейшие операции «чтения» будут использовать соединение «запись». Это гарантирует, что любые данные, записанные во время цикла запроса, могут быть немедленно обратно прочитаны из базы данных во время того же запроса. Вам решать, является ли это желаемым поведением для вашего приложения.

## # Выполнение SQL-запросов

После того как вы настроили соединение с базой данных, вы можете выполнять запросы, используя фасад `DB`. Фасад `DB` содержит методы для каждого типа запроса: `select`, `update`, `insert`, `delete`, и `statement`.

## Выполнение Select-запроса

Чтобы выполнить базовый запрос `SELECT`, вы можете использовать метод `select` фасада `DB`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать список всех пользователей приложения.
     */
    public function index(): View
    {
        $users = DB::select('select * from users where active = ? ', [1]);
```

```
        return view('user.index', ['users' => $users]);
    }
}
```

Первым аргументом, переданным методу `select`, является SQL-запрос, а вторым аргументом – любые привязки параметров, необходимые для запроса. Обычно это значения ограничений выражений `where`. Привязка параметров обеспечивает защиту от SQL-инъекций.

Метод `select` всегда возвращает «массив» результатов. Каждый результат в массиве будет объектом `stdClass` PHP, представляющим запись из базы данных:

```
use Illuminate\Support\Facades\DB;

$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}
```

## Выбор скалярных значений

Иногда ваш запрос к базе данных может вернуть единственное скалярное значение. Вместо того чтобы получать скалярный результат запроса из объекта записи, Laravel позволяет вам получать это значение напрямую с использованием метода `scalar`:

```
$burgers = DB::scalar(
    "select count(case when food = 'burger' then 1 end) as burgers from menu"
);
```

## Выбор нескольких наборов результатов

Если ваше приложение вызывает хранимые процедуры, возвращающие несколько наборов результатов, вы можете использовать метод `selectResultSets` для получения всех наборов результатов, возвращенных хранимой процедурой:

```
[$options, $notifications] = DB::selectResultSets(
    "CALL get_user_options_and_notifications(?)", $request->user()->id
```

```
);
```

## Использование именованных псевдопеременных

Вместо использования символа `?` для связывания параметров вы можете выполнить запрос, используя именованные привязки:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

## Выполнение Insert-запроса

Чтобы выполнить запрос с `INSERT`, вы можете использовать метод `insert` фасада `DB`. Как и `select`, этот метод принимает запрос SQL в качестве первого аргумента, а привязки – в качестве второго аргумента:

```
use Illuminate\Support\Facades\DB;  
  
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

## Выполнение Update-запроса

Метод `update` следует использовать для обновления существующих записей в базе данных. Количество затронутых выражением строк будут возвращены этим методом:

```
use Illuminate\Support\Facades\DB;  
  
$affected = DB::update(  
    'update users set votes = 100 where name = ?',  
    ['Anita'])  
);
```

## Выполнение Delete-запроса

Для удаления записей из базы данных следует использовать метод `delete`. Как и `update`, количество затронутых выражением строк будут возвращены этим методом:

```
use Illuminate\Support\Facades\DB;  
  
$deleted = DB::delete('delete from users');
```

## Выполнение запроса общего типа

Некоторые операторы базы данных не возвращают никакого значения. Для этих типов операций вы можете использовать метод `statement` фасада `DB`:

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

## Выполнение неподготовленного запроса

По желанию может потребоваться выполнить запрос SQL без привязки каких-либо значений. Для этого используйте метод `unprepared` фасада `DB`:

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

Поскольку неподготовленные запросы не связывают параметры, они могут быть уязвимы для SQL-инъекций. Вы никогда не должны пропускать в неподготовленное выражение значения, управляемые пользователем.

## Неявные фиксации (`implicit commit`)

При использовании в транзакциях методов `statement` и `unprepared` фасада `DB` вы должны быть осторожны, чтобы избежать операторов, которые вызывают [неявные фиксации](#). Эти операторы заставят ядро базы данных косвенно зафиксировать всю транзакцию, в результате чего Laravel не будет знать об уровне транзакции базы данных. Примером такого оператора является создание таблицы базы данных:

```
DB::unprepared('create table a (col varchar(1) null');
```

Пожалуйста, обратитесь к руководству по MySQL для ознакомления со [Списком всех операторов](#), которые выполняют неявные фиксации.

## Использование нескольких соединений к базе данных

Если ваше приложение определяет несколько соединений в конфигурационном файле `config/database.php`, то вы можете получить доступ к каждому соединению с помощью метода `connection` фасада `DB`. Имя соединения, передаваемое методу `connection`, должно соответствовать одному из подключений, перечисленных в вашем конфигурационном файле `config/database.php`, включая переопределенные с помощью глобального помощника `config` во время выполнения скрипта:

```
use Illuminate\Support\Facades\DB;  
  
$users = DB::connection('sqlite')->select(/* ... */);
```

Вы можете получить доступ к сырому, базовому экземпляру PDO текущего соединения, используя метод `getPdo` экземпляра соединения:

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

## Прослушивание событий запроса

По желанию можно указать замыкание, которое будет вызываться для каждого SQL-запроса, выполняемого вашим приложением, используя метод `listen` фасада `DB`. Этот метод может быть полезен для логирования запросов или их отладки. Вы можете зарегистрировать замыкание слушателя запросов в методе `boot` [поставщика служб](#):

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Database\Events\QueryExecuted;  
use Illuminate\Support\Facades\DB;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider
```

```

{
    /**
     * Регистрация любых служб приложения.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        DB::listen(function (QueryExecuted $query) {
            // $query->sql;
            // $query->bindings;
            // $query->time;
            // $query->toRawSql();
        });
    }
}

```

## Мониторинг общего времени выполнения запроса

Одной из обычных узких точек производительности современных веб-приложений является время, которое они затрачивают на выполнение запросов к базе данных. К счастью, Laravel может вызвать замыкание или обратный вызов по вашему выбору, когда время выполнения запросов к базе данных в течение одного запроса становится слишком велико. Для начала укажите порог времени выполнения запроса (в миллисекундах) и замыкание для метода `whenQueryingForLongerThan`. Вы можете вызвать этот метод в методе `boot` [Сервис-провайдера](#):

```

<?php

namespace App\Providers;

use Illuminate\Database\Connection;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
use Illuminate\Database\Events\QueryExecuted;

class AppServiceProvider extends ServiceProvider
{
    /**

```

```
* Register any application services.  
*/  
public function register(): void  
{  
    // ...  
}  
  
/**  
 * Bootstrap any application services.  
 */  
public function boot(): void  
{  
    DB::whenQueryingForLongerThan(500, function (Connection $connection, QueryExe  
        // Notify development team...  
    ));  
}  
}
```

## # Транзакции базы данных

Вы можете использовать метод `transaction` фасада `DB`, для выполнения набора операций в транзакции базы данных. Если при закрытии транзакции возникает исключение, то транзакция автоматически откатывается, а исключение генерируется повторно. Если замыкание выполнено успешно, то транзакция будет автоматически зафиксирована. Вам не нужно беспокоиться о ручном откате или фиксации при использовании метода `transaction`:

```
use Illuminate\Support\Facades\DB;  
  
DB::transaction(function () {  
    DB::update('update users set votes = 1');  
  
    DB::delete('delete from posts');  
});
```

## Обработка взаимоблокировок

Метод `transaction` принимает необязательный второй аргумент, который определяет, сколько раз транзакция должна быть повторена при возникновении взаимоблокировок. Как только эти попытки будут исчерпаны, будет выброшено исключение:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

## Использование транзакций вручную

Если вы хотите вручную начать транзакцию и иметь полный контроль над откатами и фиксациями, то вы можете использовать метод `beginTransaction` фасада `DB`:

```
use Illuminate\Support\Facades\DB;

DB::beginTransaction();
```

Вы можете откатить транзакцию с помощью метода `rollBack`:

```
DB::rollBack();
```

Наконец, вы можете зафиксировать транзакцию с помощью метода `commit`:

```
DB::commit();
```

Методы транзакций фасада `DB` контролируют транзакции как для [постройтеля запросов](#), так и для [Eloquent ORM](#).

## # Подключение к базе данных с помощью интерфейса командной строки Artisan

Если вы хотите подключиться к своей базе данных с помощью интерфейса командной строки, то вы можете использовать команду `db` Artisan:

```
php artisan db
```

При необходимости, вы можете указать имя соединения для подключения к базе данных, не являющееся соединением по умолчанию:

```
php artisan db mysql
```

## # Инспектирование базы данных

С помощью команд Artisan `db:show` и `db:table` вы можете получить ценную информацию о вашей базе данных и ее связанных таблицах. Для просмотра обзора вашей базы данных, включая ее размер, тип, количество открытых соединений и сводку по ее таблицам, вы можете использовать команду `db:show`:

```
php artisan db:show
```

Вы можете указать, какое соединение с базой данных следует использовать, передав имя соединения с помощью опции `--database`:

```
php artisan db:show --database=pgsql
```

Если вы хотите включить количество строк в таблицах и подробности о представлениях базы данных в выводе команды, вы можете указать соответственно опции `--counts` и `--views`. На больших базах данных получение количества строк и сведений о представлениях может занять много времени:

```
php artisan db:show --counts --views
```

Кроме того, вы можете использовать следующие методы `Schema` для проверки вашей базы данных:

```
use Illuminate\Support\Facades\Schema;
```

```
$tables = Schema::getTables();
```

```
$views = Schema::getViews();
$columns = Schema::getColumns('users');
$indexes = Schema::getIndexes('users');
$foreignKeys = Schema::getForeignKeys('users');
```

Если вы хотите проверить соединение с базой данных, которое не является соединением вашего приложения по умолчанию, вы можете использовать метод `connection`:

```
$columns = Schema::connection('sqlite')->getColumns('users');
```

## Обзор таблиц

Если вы хотите получить обзор отдельной таблицы в вашей базе данных, вы можете выполнить команду Artisan `db:table`. Эта команда предоставляет общий обзор таблицы базы данных, включая ее столбцы, типы, атрибуты, ключи и индексы:

```
php artisan db:table users
```

## # Мониторинг баз данных

Используя команду Artisan `db:monitor`, вы можете поручить Laravel отправить `Illuminate\Database\Events\DatabaseBusy`, если ваша база данных управляет большим количеством открытых соединений, чем задано.

Для начала вам следует запланировать выполнение команды `db:monitor` каждую минуту. Команда принимает имена конфигураций подключений к базе данных, которые вы хотите мониторить, а также максимальное количество открытых соединений, которые допустимы до отправки события:

```
php artisan db:monitor --databases=mysql,pgsql --max=100
```

Одного планирования этой команды недостаточно для отправки уведомления о количестве открытых соединений. Когда команда обнаруживает базу данных с количеством открытых соединений, превышающим ваш порог, будет отправлено событие `DatabaseBusy`. Вы должны прослушивать это событие в файле

`AppServiceProvider` вашего приложения, чтобы отправить уведомление вам или вашей команде разработки:

```
use App\Notifications\DatabaseApproachingMaxConnections;
use Illuminate\Database\Events\DatabaseBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * Запуск любых служб приложения.
 */
public function boot(): void
{
    Event::listen(function (DatabaseBusy $event) {
        Notification::route('mail', 'dev@example.com')
            ->notify(new DatabaseApproachingMaxConnections(
                $event->connectionName,
                $event->connections
            ));
    });
}
```

# Построитель запросов

# Введение

# Выполнение запросов к базе данных

# Разбиение результатов

# Отложенная потоковая передача результатов

# Агрегатные функции

# Выражения Select

# Сырые SQL-выражения

# Сырые sql-выражения

# Соединения Joins

# Объединения результатов Unions

# Основные выражения Where

# Выражения Where

# Выражения Or Where

# Выражение Where Not

# Выражения Where Any / All / None

# Выражения Where и JSON

# Дополнительные выражения Where

# Логическая группировка

# Расширенные выражения Where

# Выражения Where Exists

# Подзапросы выражений Where

# Полнотекстовый поиск

# Сортировка, группировка, ограничение и смещение

# Сортировка

# Группировка

# Ограничение и смещение

# Условные выражения

# Вставка

# Обновления-вставки

## # Обновление

- # Обновление столбцов JSON
- # Увеличение и уменьшение отдельных значений

## # Удаление

- # Пессимистическая блокировка
- # Отладка

# # Введение

Построитель запросов к базе данных Laravel предлагает удобный и гибкий интерфейс для создания и выполнения запросов к базе данных. Его можно использовать для выполнения большинства операций с базой данных в вашем приложении и он отлично работает со всеми поддерживаемыми Laravel системами баз данных.

Построитель запросов Laravel использует связывание параметров PDO для защиты приложения от SQL-инъекций. Нет необходимости чистить строки, передаваемые как связываемые параметры.

PDO не поддерживает связывание имен столбцов. Поэтому, вы никогда не должны использовать какие-либо входящие от пользователя данные в качестве имен столбцов, используемые вашими запросами, включая столбцы в запросах `order by` и т.д.

# # Выполнение запросов к базе данных

## Получение всех строк из таблицы

Вы можете использовать метод `table` фасада `DB`, чтобы начать запрос. Метод `table` возвращает текущий экземпляр построителя запросов для данной таблицы, позволяя вам связать больше ограничений к запросу и, наконец, получить результаты, используя метод `get`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать список всех пользователей приложения.
     */
    public function index(): View
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

Метод `get` возвращает экземпляр `Illuminate\Support\Collection`, содержащий результаты запроса, где каждый результат является экземпляром объекта `stdClass`. Вы можете получить доступ к значению каждого столбца, обратившись к столбцу как к свойству объекта:

```
use Illuminate\Support\Facades\DB;

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

foreach ($users as $user) {
    echo $user->name;
}
```

Коллекции Laravel содержат множество чрезвычайно мощных методов для работы с наборами данных. Для получения дополнительной информации о коллекциях Laravel ознакомьтесь с [их документацией](#).

## Получение одной строки / столбца из таблицы

Если вам просто нужно получить одну строку из таблицы базы данных, вы можете использовать метод `first` фасада `DB`. Этот метод вернет единственный объект `stdClass`:

```
$user = DB::table('users')->where('name', 'John')->first();  
  
return $user->email;
```

Если вы хотите получить одну строку из таблицы базы данных, но получаете `Illuminate\Database\RecordNotFoundException`, если соответствующая строка не найдена, вы можете использовать метод `firstOrFail`. Если `RecordNotFoundException` не перехвачен, HTTP-ответ 404 автоматически отправляется обратно клиенту:

```
$user = DB::table('users')->where('name', 'John')->firstOrFail();
```

Если вам не нужна вся строка, вы можете извлечь одно значение из записи с помощью метода `value`. Этот метод вернет значение столбца напрямую:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Чтобы получить одну строку по значению столбца `id`, используйте метод `find`:

```
$user = DB::table('users')->find(3);
```

## Получение списка значений столбца

Если вы хотите получить экземпляр `Illuminate\Support\Collection`, содержащий значения одного столбца, вы можете использовать метод `pluck`. В этом примере мы получим коллекцию из названий пользователей:

```
use Illuminate\Support\Facades\DB;  
  
$titles = DB::table('users')->pluck('title');  
  
foreach ($titles as $title) {
```

```
    echo $title;  
}
```

Вы можете указать столбец, который результирующая коллекция должна использовать в качестве ключей, указав второй аргумент методу `pluck`:

```
$titles = DB::table('users')->pluck('title', 'name');  
  
foreach ($titles as $name => $title) {  
    echo $title;  
}
```

## Разбиение результатов

Если вам нужно работать с тысячами записей базы данных, рассмотрите возможность использования метода `chunk` фасада `DB`. Этот метод извлекает за раз небольшой фрагмент результатов и передает каждый фрагмент в функцию-аргумент для обработки. Например, давайте извлечем всю таблицу `users` фрагментами по 100 записей за раз:

```
use Illuminate\Support\Collection;  
use Illuminate\Support\Facades\DB;  
  
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {  
    foreach ($users as $user) {  
        // ...  
    }  
});
```

Вы можете остановить обработку последующих фрагментов, вернув из функции обработки `false`:

```
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {  
    // Обрабатываем записи ...  
  
    return false;  
});
```

Если вы обновляете записи базы данных во время фрагментирования результатов, то результаты ваших фрагментов могут измениться неожиданным образом. Если

вы планируете обновлять полученные записи при фрагментировании, всегда лучше использовать вместо этого метод `chunkById`. Этот метод автоматически разбивает результаты на фрагменты на основе первичного ключа записи:

```
DB::table('users')->where('active', false)
->chunkById(100, function (Collection $users) {
    foreach ($users as $user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    }
});
```

Поскольку методы `chunkById` и `lazyById` добавляют свои собственные условия "where" к выполняемому запросу, вам обычно следует логически группировать свои собственные условия внутри замыкания:

```
DB::table('users')->where(function ($query) {
    $query->where('credits', 1)->orWhere('credits', 2);
})->chunkById(100, function (Collection $users) {
    foreach ($users as $user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['credits' => 3]);
    }
});
```

При обновлении или удалении записей внутри функции-аргумента, любые изменения первичного или внешних ключей могут повлиять на запрос очередного фрагмента. Это может потенциально привести к тому, что записи могут не быть включены в последующие результаты выполнения функции.

## Отложенная потоковая передача результатов

Метод `lazy` работает аналогично [методу `chunk`](#) в том смысле, что он выполняет запрос по частям. Однако вместо передачи каждого фрагмента непосредственно в функцию-обработчик, метод `lazy()` возвращает экземпляр [LazyCollection](#), что позволяет вам взаимодействовать с результатами как с единым потоком:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function (object $user) {
    // ...
});
```

Еще раз, если вы планируете обновлять полученные записи во время их итерации, лучше вместо этого использовать методы `lazyById` или `lazyByIdDesc`. Эти методы автоматически разбивают результаты «постранично» на основе первичного ключа записи:

```
DB::table('users')->where('active', false)
->lazyById()->each(function (object $user) {
    DB::table('users')
        ->where('id', $user->id)
        ->update(['active' => true]);
});
```

При обновлении или удалении записей во время их итерации любые изменения первичного ключа или внешних ключей могут повлиять на запрос фрагмента. Это может потенциально привести к тому, что записи не будут включены в результирующий набор.

## Агрегатные функции

Построитель запросов также содержит множество методов для получения агрегированных значений, таких как `count`, `max`, `min`, `avg`, и `sum`. После создания запроса вы можете вызвать любой из этих методов:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Конечно, вы можете комбинировать эти методы с другими выражениями, чтобы уточнить способ вычисления вашего совокупного значения:

```
$price = DB::table('orders')
    ->where('finalized', 1)
    ->avg('price');
```

## Определение наличия записей

Вместо использования метода `count` для определения существования каких-либо записей, соответствующих ограничениям вашего запроса, используйте методы `exists` и `doesntExist`:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

## # Выражения Select

### Уточнения выражения Select

Возможно, вам не всегда нужно выбирать все столбцы из таблицы базы данных. Используя метод `select`, вы можете указать собственное выражение `SELECT` для запроса:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
```

```
->select('name', 'email as user_email')  
->get();
```

Метод `distinct` позволяет вам заставить запрос возвращать уникальные результаты:

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

Если у вас уже есть экземпляр построителя запросов, и вы хотите добавить столбец к существующему выражению `SELECT`, то вы можете использовать метод `addSelect`:

```
$query = DB::table('users')->select('name');
```

```
$users = $query->addSelect('age')->get();
```

## # Сырые SQL-выражения

Иногда вам может понадобиться вставить в запрос произвольную строку, содержащую часть SQL-запроса. Для этого вы можете использовать метод `raw` фасада `DB`:

```
$users = DB::table('users')  
    ->select(DB::raw('count(*) as user_count, status'))  
    ->where('status', '<>', 1)  
    ->groupBy('status')  
    ->get();
```

Сырые выражения будут вставлены в запрос в виде строк, поэтому следует проявлять особую осторожность, чтобы не создавать уязвимости для SQL-инъекций.

## Сырые sql-выражения

Вместо использования метода `DB::raw`, вы также можете использовать следующие методы для вставки произвольного SQL-выражения в различные части вашего запроса. **Помните, Laravel не может гарантировать, что любой запрос, использующий сырые SQL-выражения, защищен от уязвимостей SQL-инъекций.**

## selectRaw

Метод `selectRaw` можно использовать вместо `addSelect(DB::raw(/* ... */))`. Этот метод принимает необязательный массив параметров для подстановки в качестве второго аргумента:

```
$orders = DB::table('orders')
    ->selectRaw('price * ? as price_with_tax', [1.0825])
    ->get();
```

## whereRaw / orWhereRaw

Методы `whereRaw` и `orWhereRaw` можно использовать для вставки сырого SQL-выражения `WHERE` в ваш запрос. Эти методы принимают необязательный массив параметров в качестве второго аргумента:

```
$orders = DB::table('orders')
    ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
    ->get();
```

## havingRaw / orHavingRaw

Методы `havingRaw` и `orHavingRaw` могут использоваться для вставки необработанной строки в качестве значения выражения `HAVING`. Эти методы принимают необязательный массив параметров в качестве второго аргумента:

```
$orders = DB::table('orders')
    ->select('department', DB::raw('SUM(price) as total_sales'))
    ->groupBy('department')
    ->havingRaw('SUM(price) > ?', [2500])
    ->get();
```

## orderByRaw

Метод `orderByRaw` используется для предоставления необработанной строки в качестве значения выражения `ORDER BY`:

```
$orders = DB::table('orders')
    ->orderByRaw('updated_at - created_at DESC')
    ->get();
```

## groupByRaw

Метод `groupByRaw` используется для предоставления необработанной строки в качестве значения выражения `GROUP BY`:

```
$orders = DB::table('orders')
    ->select('city', 'state')
    ->groupByRaw('city, state')
    ->get();
```

# # Соединения Joins

## Inner Join

Построитель запросов также может использоваться для добавления выражений соединения (`join`) к вашим запросам. Чтобы выполнить базовое «внутреннее соединение» (inner join), вы можете использовать метод `join`. Первым аргументом, передаваемым методу `join`, является имя таблицы, к которой вам нужно присоединиться, а остальные аргументы определяют ограничения столбца для соединения. Вы даже можете соединить несколько таблиц в один запрос:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

## Left Join / Right Join

Если вы хотите выполнить «левое соединение» или «правое соединение» вместо «внутреннего соединения», используйте методы `leftJoin` или `rightJoin`. Эти методы имеют ту же сигнатуру, что и метод `join`:

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();

$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

## Cross Join

Вы можете использовать метод `crossJoin` для выполнения «перекрестного соединения». Перекрестные соединения генерируют декартово произведение между первой таблицей и соединяемой таблицей:

```
$sizes = DB::table('sizes')
    ->crossJoin('colors')
    ->get();
```

## Расширенные выражения соединения

Вы также можете указать более сложные выражения соединения. Для начала передайте функцию в качестве второго аргумента методу `join`. Функция получит экземпляр `Illuminate\Database\Query\JoinClause`, который позволяет вам указать ограничения `JOIN`:

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */);
    })
    ->get();
```

Если вы хотите использовать выражение `WHERE` в своих соединениях, вы можете использовать методы `where` и `orWhere` экземпляра `JoinClause`. Вместо сравнения двух столбцов эти методы будут сравнивать столбец со значением:

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')
        ->where('contacts.user_id', '>', 5);
    })
    ->get();
```

## Подзапросы соединений

Вы можете использовать методы `joinSub`, `leftJoinSub`, и `rightJoinSub`, чтобы присоединить запрос к подзапросу. Каждый из этих методов получает три аргумента: подзапрос, псевдоним таблицы и функцию, определяющую связанные столбцы. В этом примере мы получим коллекцию пользователей, где каждая запись пользователя также содержит временную метку `created_at` последнего опубликованного поста пользователя в блоге:

```
$latestPosts = DB::table('posts')
    ->select('user_id', DB::raw('MAX(created_at) as last_post_created'))
    ->where('is_published', true)
    ->groupBy('user_id');

$users = DB::table('users')
    ->joinSub($latestPosts, 'latest_posts', function (JoinClause $join) {
        $join->on('users.id', '=', 'latest_posts.user_id');
    })->get();
```

## Боковые соединения (Lateral Joins)

Боковые соединения в настоящее время поддерживаются PostgreSQL, MySQL >= 8.0.14 и SQL Server. Вы можете использовать методы `joinLateral` и `leftJoinLateral` для выполнения “бокового соединения” с подзапросом. Каждый из этих методов принимает два аргумента: подзапрос и его псевдоним таблицы. Условие(я) соединения должно быть указано в `where` выражении данного подзапроса. Боковые соединения оцениваются

для каждой строки и могут ссылаться на столбцы вне подзапроса.

В этом примере мы получим коллекцию пользователей, а также три последних блог-поста пользователя. Для каждого пользователя может быть до трех строк в наборе результатов: по одной для каждого из его последних блог-постов. Условие соединения указывается с помощью `whereColumn` выражения внутри подзапроса, ссылаясь на текущую строку пользователя:

```
$latestPosts = DB::table('posts')
    ->select('id as post_id', 'title as post_title', 'created_at as post_created_at')
    ->whereColumn('user_id', 'users.id')
    ->orderBy('created_at', 'desc')
    ->limit(3);

$users = DB::table('users')
    ->joinLateral($latestPosts, 'latest_posts')
    ->get();
```

## # Объединения результатов Unions

Построитель запросов также содержит удобный метод «объединения» двух или более запросов вместе. Например, вы можете создать первый запрос и использовать метод `union` для объединения его с другими запросами:

```
use Illuminate\Support\Facades\DB;

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

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

В дополнение к методу `union`, построитель запросов содержит метод `unionAll`. Запросы, объединенные с использованием метода `unionAll`, не будут удалять

повторяющиеся результаты. Метод `unionAll` имеет ту же сигнатуру, что и метод `union`.

## # Основные выражения Where

### Выражения Where

Вы можете использовать метод `where` построителя запросов, чтобы добавить в запрос выражения `WHERE`. Самый простой вызов метода `where` требует трех аргументов. Первый аргумент – это имя столбца. Второй аргумент – это оператор, который может быть любым из поддерживаемых базой данных операторов. Третий аргумент – это значение, которое нужно сравнить со значением столбца.

Например, следующий запрос извлекает пользователей, у которых значение столбца `votes` равно `100`, а значение столбца `age` больше, чем `35`:

```
$users = DB::table('users')
    ->where('votes', '=', 100)
    ->where('age', '>', 35)
    ->get();
```

Для удобства, если вы хотите убедиться, что столбец соответствует `=` переданному значению, то вы можете передать это значение в качестве второго аргумента в метод `where`. Laravel будет предполагать, что вы хотите использовать оператор `=:`

```
$users = DB::table('users')->where('votes', 100)->get();
```

Как упоминалось ранее, вы можете использовать любой оператор, который поддерживается вашей системой баз данных:

```
$users = DB::table('users')
    ->where('votes', '>=', 100)
    ->get();
```

```
$users = DB::table('users')
    ->where('votes', '<>', 100)
    ->get();
```

```
$users = DB::table('users')
```

```
->where('name', 'like', 'T%')
->get();
```

Вы также можете передать массив условий методу `where`. Каждый элемент массива должен быть массивом, содержащим три аргумента, как и обычно передаваемых методу `where`:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

PDO не поддерживает привязку имен столбцов. Поэтому вы никогда не должны брать из пользовательского ввода имена столбцов для совершения запросов, включая столбцы "order by".

MySQL и MariaDB автоматически преобразуют строки в целые числа при сравнении чисел-строк. В этом процессе нечисловые строки преобразуются в `0`, что может привести к неожиданным результатам. Например, если в вашей таблице есть столбец `secret` со значением `aaa` и вы запускаете `User::where('secret', 0)`, будет возвращена эта строка. Чтобы избежать этого, убедитесь, что все значения приведены к соответствующим типам, прежде чем использовать их в запросах.

## Выражения Or Where

При объединении в цепочку вызовов метода `where` построителя запросов выражения `WHERE` будут объединены вместе с помощью оператора `AND`. Однако, вы можете использовать метод `orWhere` для добавления выражения к запросу

с помощью оператора `OR`. Метод `orWhere` принимает те же аргументы, что и метод `where`:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

Если вам нужно сгруппировать условие `OR` в круглых скобках, вы можете передать функцию в качестве первого аргумента методу `orWhere`:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere(function (Builder $query) {
        $query->where('name', 'Abigail')
            ->where('votes', '>', 50);
    })
    ->get();
```

В приведенном выше примере будет получен следующий SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

Вы всегда должны группировать вызовы `orWhere`, чтобы избежать неожиданного поведения при применении [глобальных диапазонов](#).

## Выражение Where Not

Методы `whereNot` и `orWhereNot` могут использоваться для отрицания заданной группы ограничений запроса. Например, в следующем запросе исключаются товары, находящиеся на распродаже или имеющие цену менее десяти:

```
$products = DB::table('products')
    ->whereNot(function (Builder $query) {
        $query->where('clearance', true)
```

```
->orWhere('price', '<', 10);
})
->get();
```

## Выражения Where Any / All / None

Иногда вам может понадобиться применить одни и те же условия к нескольким столбцам запроса. Например, вы можете хотеть выбрать все записи, где хотя бы один столбец из списка соответствует определенному значению. Это можно сделать с помощью метода `whereAny`:

```
$users = DB::table('users')
    ->where('active', true)
    ->whereAny([
        'name',
        'email',
        'phone',
    ], 'like', 'Example%')
    ->get();
```

Запрос выше приведет к следующему SQL:

```
SELECT *
FROM users
WHERE active = true AND (
    name LIKE 'Example%' OR
    email LIKE 'Example%' OR
    phone LIKE 'Example%'
)
```

Аналогично метод `whereAll` может быть использован для извлечения записей, где все указанные столбцы соответствуют заданному условию:

```
$posts = DB::table('posts')
    ->where('published', true)
    ->whereAll([
        'title',
        'content',
    ], 'like', '%Laravel%')
    ->get();
```

Запрос выше приведет к следующему SQL:

```
SELECT *
FROM posts
WHERE published = true AND (
    title LIKE '%Laravel%' AND
    content LIKE '%Laravel%'
)
```

Метод `whereNone` можно использовать для извлечения записей, в которых ни один из заданных столбцов не соответствует заданному ограничению:

```
$posts = DB::table('albums')
    ->where('published', true)
    ->whereNone([
        'title',
        'lyrics',
        'tags',
    ], 'like', '%explicit%')
    ->get();
```

Результатом приведенного выше запроса будет следующий SQL:

```
SELECT *
FROM albums
WHERE published = true AND NOT (
    title LIKE '%explicit%' OR
    lyrics LIKE '%explicit%' OR
    tags LIKE '%explicit%'
)
```

## Выражения Where и JSON

Laravel также поддерживает запросы к типам столбцов JSON в базах данных, которые предоставляют поддержку для типов столбцов JSON. В настоящее время это включает MariaDB 10.3+, MySQL 8.0+, PostgreSQL 12.0+, SQL Server 2017+ и SQLite 3.39.0. Для выполнения запроса к столбцу JSON используйте оператор `->`:

```
$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
```

```
->get();
```

Вы можете использовать `whereJsonContains` для запроса массивов JSON.

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', 'en')
    ->get();
```

Если ваше приложение использует базы данных MariaDB, MySQL или PostgreSQL, вы можете передать массив значений методу `whereJsonContains`:

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', ['en', 'de'])
    ->get();
```

Вы можете использовать метод `whereJsonLength` для запроса массивов JSON по их длине:

```
$users = DB::table('users')
    ->whereJsonLength('options->languages', 0)
    ->get();

$users = DB::table('users')
    ->whereJsonLength('options->languages', '>', 1)
    ->get();
```

## Дополнительные выражения Where

### `whereLike / orWhereLike / whereNotLike / orWhereNotLike`

Метод `whereLike` позволяет добавлять в запрос предложения “LIKE” для сопоставления с образцом. Эти методы обеспечивают независимый от базы данных способ выполнения запросов на сопоставление строк с возможностью переключения чувствительности к регистру. По умолчанию сопоставление строк не учитывает регистр:

```
$users = DB::table('users')
    ->whereLike('name', '%John%')
```

```
->get();
```

Вы можете включить поиск с учетом регистра с помощью аргумента `caseSensitive`:

```
$users = DB::table('users')
    ->whereLike('name', '%John%', caseSensitive: true)
    ->get();
```

Метод `orWhereLike` позволяет добавить предложение “or” с условием LIKE:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhereLike('name', '%John%')
    ->get();
```

Метод `whereNotLike` позволяет добавлять в запрос предложения “NOT LIKE”:

```
$users = DB::table('users')
    ->whereNotLike('name', '%John%')
    ->get();
```

Аналогичным образом вы можете использовать `orWhereNotLike` для добавления предложения “or” с условием NOT LIKE:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhereNotLike('name', '%John%')
    ->get();
```

Параметр поиска `whereLike` с учетом регистра в настоящее время не поддерживается на SQL Server.

**whereIn / whereNotIn / orWhereIn / orWhereNotIn**

Метод `whereIn` проверяет, что значение переданного столбца содержится в указанном массиве:

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

Метод `whereNotIn` проверяет, что значение переданного столбца не содержится в указанном массиве:

```
$users = DB::table('users')
    ->whereNotIn('id', [1, 2, 3])
    ->get();
```

Вы также можете использовать объект запроса в качестве второго аргумента метода `whereIn`:

```
$activeUsers = DB::table('users')->select('id')->where('is_active', 1);

$users = DB::table('comments')
    ->whereIn('user_id', $activeUsers)
    ->get();
```

Приведенный выше пример создаст следующий SQL-запрос:

```
select * from comments where user_id in (
    select id
    from users
    where is_active = 1
)
```

Если вы добавляете в свой запрос большой массив связываемых целочисленных параметров, то методы `whereIntegerInRaw` или `whereIntegerNotInRaw` могут использоваться для значительного сокращения потребляемой памяти.

## **whereBetween / orWhereBetween**

Метод `whereBetween` проверяет, что значение столбца находится между двумя значениями:

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])
    ->get();
```

## **whereNotBetween / orWhereNotBetween**

Метод `whereNotBetween` проверяет, что значение столбца находится за пределами двух значений:

```
$users = DB::table('users')
    ->whereNotBetween('votes', [1, 100])
    ->get();
```

## **whereBetweenColumns / whereNotBetweenColumns / orWhereBetweenColumns / orWhereNotBetweenColumns**

Метод `whereBetweenColumns` проверяет, что значение столбца находится между двумя значениями двух столбцов в одной строке таблицы:

```
$patients = DB::table('patients')
    ->whereBetweenColumns('weight', ['minimum_allowed_weight', 'maximum_allowed_weight'])
    ->get();
```

Метод `whereNotBetweenColumns` проверяет, что значение столбца находится за пределами двух значений двух столбцов в одной строке таблицы:

```
$patients = DB::table('patients')
    ->whereNotBetweenColumns('weight', ['minimum_allowed_weight', 'maximum_allowed_weight'])
    ->get();
```

## **whereNull / whereNotNull / orWhereNull / orWhereNotNull**

Метод `whereNull` проверяет, что значение переданного столбца равно `NULL`:

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

Метод `whereNotNull` проверяет, что значение переданного столбца не равно `NULL`:

```
$users = DB::table('users')
    ->whereNotNull('updated_at')
    ->get();
```

### **whereDate / whereMonth / whereDay / whereYear / whereTime**

Метод `whereDate` используется для сравнения значения столбца с датой:

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-12-31')
    ->get();
```

Метод `whereMonth` используется для сравнения значения столбца с конкретным месяцем:

```
$users = DB::table('users')
    ->whereMonth('created_at', '12')
    ->get();
```

Метод `whereDay` используется для сравнения значения столбца с определенным днем месяца:

```
$users = DB::table('users')
    ->whereDay('created_at', '31')
    ->get();
```

Метод `whereYear` используется для сравнения значения столбца с конкретным годом:

```
$users = DB::table('users')
    ->whereYear('created_at', '2016')
    ->get();
```

Метод `whereTime` используется для сравнения значения столбца с определенным временем:

```
$users = DB::table('users')
    ->whereTime('created_at', '=', '11:20:45')
    ->get();
```

## whereColumn / orWhereColumn

Метод `whereColumn` используется для проверки равенства двух столбцов:

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name')
    ->get();
```

Вы также можете передать оператор сравнения методу `whereColumn`:

```
$users = DB::table('users')
    ->whereColumn('updated_at', '>', 'created_at')
    ->get();
```

Вы также можете передать массив сравнений столбцов методу `whereColumn`. Эти условия будут объединены с помощью оператора `AND`:

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', '=', 'last_name'],
        ['updated_at', '>', 'created_at'],
    ])->get();
```

## Логическая группировка

Иногда требуется сгруппировать несколько выражений `WHERE` в круглых скобках, чтобы добиться желаемой логической группировки вашего запроса. Фактически, вы должны всегда группировать вызовы метода `orWhere` в круглых скобках, чтобы избежать неожиданного поведения запроса. Для этого вы можете передать функцию методу `where`:

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

Как вы можете видеть, передача функции в метод `where` инструктирует построитель запросов начать группу ограничений. Функция получит экземпляр построителя запросов, который вы можете использовать для задания ограничений, которые должны содержаться в группе скобок. В приведенном выше примере будет получен следующий SQL:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

Вы всегда должны группировать вызовы `orWhere`, чтобы избежать неожиданного поведения при применении [глобальных диапазонов](#).

## Расширенные выражения Where

### Выражения Where Exists

Метод `whereExists` позволяет писать выражения `WHERE EXISTS` SQL. Метод `whereExists` принимает функцию, которая получит экземпляр построителя запросов, позволяя вам определить запрос, который должен быть помещен внутри выражения `EXISTS`:

```
$users = DB::table('users')
    ->whereExists(function (Builder $query) {
```

```
$query->select(DB::raw(1))
    ->from('orders')
    ->whereColumn('orders.user_id', 'users.id');
})
->get();
```

Кроме того, вы можете предоставить объект запроса методу `whereExists` вместо замыкания:

```
$orders = DB::table('orders')
->select(DB::raw(1))
->whereColumn('orders.user_id', 'users.id');

$users = DB::table('users')
->whereExists($orders)
->get();
```

Оба приведенных выше примера создадут следующий SQL-запрос:

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

## Подзапросы выражений Where

Иногда требуется создать выражение `WHERE`, которое сравнивает результаты подзапроса с переданным значением. Вы можете добиться этого, передав функцию и значение методу `where`. Например, следующий запрос будет извлекать всех пользователей, недавно имевших «членство» указанного типа:

```
use App\Models\User;
use Illuminate\Database\Query\Builder;

$users = User::where(function (Builder $query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
```

```
->limit(1);  
}, 'Pro')->get();
```

Или вам может потребоваться создать выражение “where”, которое сравнивает столбец с результатами подзапроса. Вы можете сделать это, передав методу `where` столбец, оператор и функцию. Например, следующий запрос будет извлекать все записи о доходах, где сумма меньше средней:

```
use App\Models\Income;  
  
$incomes = Income::where('amount', '<', function (Builder $query) {  
    $query->selectRaw('avg(i.amount)')->from('incomes as i');  
})->get();
```

## Полнотекстовый поиск

Полнотекстовый поиск поддерживаются в настоящее время для MariaDB, MySQL и PostgreSQL.

Методы `whereFullText` и `orWhereFullText` позволяют добавлять полнотекстовые “условия” в запрос для столбцов, имеющих [полнотекстовые индексы](#). Laravel автоматически преобразует эти методы в соответствующий SQL-код для используемой базы данных. Например, для приложений, использующих MariaDB или MySQL, будет сгенерировано условие `MATCH AGAINST`:

```
$users = DB::table('users')  
->whereFullText('bio', 'web developer')  
->get();
```

## # Сортировка, группировка, ограничение и смещение

### Сортировка

## Метод orderBy

Метод `orderBy` позволяет вам сортировать результаты запроса по конкретному столбцу. Первый аргумент, принимаемый методом `orderBy`, должен быть столбцом, по которому вы хотите выполнить сортировку, а второй аргумент определяет направление сортировки и может быть либо `asc`, либо `desc`:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->get();
```

Для сортировки по нескольким столбцам вы можете просто вызывать `orderBy` столько раз, сколько необходимо:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->orderBy('email', 'asc')
    ->get();
```

## Методы latest и oldest

Методы `latest` и `oldest` позволяют легко упорядочивать результаты по дате. По умолчанию результат будет упорядочен по столбцу `created_at` таблицы. Или вы можете передать имя столбца, по которому хотите сортировать:

```
$user = DB::table('users')
    ->latest()
    ->first();
```

## Случайный порядок

Метод `inRandomOrder` используется для случайной сортировки результатов запроса. Например, вы можете использовать этот метод для выборки случайного пользователя:

```
$randomUser = DB::table('users')
    ->inRandomOrder()
    ->first();
```

## Удаление существующих сортировок

Метод `reorder` удаляет все выражения `ORDER BY`, которые ранее были применены к запросу:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

Вы можете передать столбец и направление при вызове метода `reorder`, чтобы удалить все существующие выражения `ORDER BY` и применить к запросу совершенно новый порядок:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

## Группировка

### Методы `groupBy` и `having`

Как и следовало ожидать, для группировки результатов запроса могут использоваться методы `groupBy` и `having`. Сигнатура метода `having` аналогична сигнатуре метода `where`:

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

Вы можете использовать метод `havingBetween` для фильтрации результатов в заданном диапазоне:

```
$report = DB::table('orders')
    ->selectRaw('count(id) as number_of_orders, customer_id')
    ->groupBy('customer_id')
    ->havingBetween('number_of_orders', [5, 15])
    ->get();
```

Вы можете передать несколько аргументов методу `groupBy` для группировки по нескольким столбцам:

```
$users = DB::table('users')
    ->groupBy('first_name', 'status')
    ->having('account_id', '>', 100)
    ->get();
```

Чтобы создать более сложные операторы `having`, см. метод [havingRaw](#).

## Ограничение и смещение

### Методы `skip` и `take`

Вы можете использовать методы `skip` и `take`, чтобы ограничить количество результатов, возвращаемых запросом, или пропустить указанное количество результатов из запроса:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Как вариант, вы можете использовать методы `limit` и `offset`. Эти методы функционально эквивалентны методам `take` и `skip` соответственно:

```
$users = DB::table('users')
    ->offset(10)
    ->limit(5)
    ->get();
```

## # Условные выражения

Иногда может потребоваться, чтобы определенные выражения запроса применялись к запросу на основании другого условия. Например, бывает необходимо применить оператор `WHERE` только в том случае, если переданное входящее значение присутствует в HTTP-запросе. Вы можете сделать это с помощью метода `when`:

```
$role = $request->input('role');

$users = DB::table('users')
    ->when($role, function (Builder $query, string $role) {
        $query->where('role_id', $role);
    })
    ->get();
```

Метод `when` выполняет переданную функцию-аргумент только тогда, когда первый аргумент равен `true`. Если первый аргумент – `false`, функция не будет выполнена. Итак, в приведенном выше примере функция метода `when` будет вызываться только в том случае, если поле `role` присутствует во входящем запросе и оценивается как `true`.

Вы можете передать другую функцию в качестве третьего аргумента методу `when`. Это функция будет выполнена только в том случае, если первый аргумент оценивается как `false`. Чтобы проиллюстрировать этот функционал, определим порядок вывода записей по умолчанию для запроса:

```
$sortByVotes = $request->boolean('sort_by_votes');

$users = DB::table('users')
    ->when($sortByVotes, function (Builder $query, bool $sortByVotes) {
        $query->orderBy('votes');
    }, function (Builder $query) {
        $query->orderBy('name');
    })
    ->get();
```

## # Вставка

Построитель запросов также содержит метод `insert`, который можно использовать для вставки записей в таблицу базы данных. Метод `insert` принимает массив имен и значений столбцов:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

Вы можете вставить сразу несколько записей, передав массив массивов. Каждый из массивов представляет собой запись, которую нужно вставить в таблицу:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

Метод `insertOrIgnore` позволяет игнорировать ошибки при вставке записей в базу данных. При использовании этого метода следует помнить, что ошибки дублирования записей будут проигнорированы, и другие виды ошибок также могут быть проигнорированы в зависимости от используемой базы данных. Например, `insertOrIgnore` пропускает [строгий режим MySQL](#):

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

Метод `insertUsing` вставляет новые записи в таблицу, используя подзапрос для определения данных, которые должны быть вставлены:

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
)->where('updated_at', '<=', now()->subMonth()));
```

## Автоинкрементирование идентификаторов

Если таблица имеет автоинкрементный идентификатор, то используйте метод `insertGetId`, чтобы вставить запись и затем получить идентификатор этой записи:

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

При использовании PostgreSQL метод `insertGetId` ожидает, что автоинкрементный столбец будет называться `id`. Если вы хотите получить идентификатор из другой «последовательности», вы можете передать имя столбца в качестве второго параметра методу `insertGetId`.

## Обновления-вставки

Метод `upsert` вставляет записи, которые не существуют, и обновляет записи, которые уже существуют, новыми значениями, которые вы можете указать. Первый аргумент метода состоит из значений для вставки или обновления, а второй аргумент перечисляет столбцы, которые однозначно идентифицируют записи в связанной таблице. Третий и последний аргумент метода – это массив столбцов, который следует обновить, если соответствующая запись уже существует в базе данных:

```
DB::table('flights')->upsert(  
    [  
        ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],  
        ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]  
    ],  
    ['departure', 'destination'],  
    ['price']  
);
```

В приведенном выше примере Laravel попытается вставить две записи. Если запись уже существует с такими же значениями столбцов `departure` и `destination`, то Laravel обновит столбец `price` этой записи.

Все базы данных, кроме SQL Server, требуют, чтобы столбцы во втором аргументе метода `upsert` имели «первичный» или «уникальный» индекс. Вдобавок, драйверы базы данных MariaDB и MySQL игнорируют второй аргумент метода `upsert` и всегда использует

«первичный» и «уникальный» индексы таблицы для обнаружения существующих записей.

## # Обновление

Помимо вставки записей в базу данных, построитель запросов также может обновлять существующие записи с помощью метода `update`. Метод `update`, как и метод `insert`, принимает массив пар столбцов и значений, указывающих столбцы, которые нужно обновить. Вы можете ограничить запрос `update` с помощью выражений `WHERE`:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);
```

## Обновление или вставка

Иногда требуется обновить существующую запись в базе данных или создать ее, если соответствующей записи не существует. В этом сценарии может использоваться метод `updateOrInsert`. Метод `updateOrInsert` принимает два аргумента: массив условий, по которым нужно найти запись, и массив пар столбцов и значений, указывающих столбцы, которые нужно обновить.

Метод `updateOrInsert` попытается найти соответствующую запись в базе данных, используя пары столбец и значение первого аргумента. Если запись существует, она будет обновлена значениями второго аргумента. Если запись не может быть найдена, будет вставлена новая запись с объединенными атрибутами обоих аргументов:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

You may provide a closure to the `updateOrInsert` method to customize the attributes that are updated or inserted into the database based on the existence of a matching record:

Вы можете предоставить закрытие метода `updateOrInsert`, чтобы настроить атрибуты, которые обновляются или вставляются в базу данных на основе существования соответствующей записи:

```
DB::table('users')->updateOrInsert(  
    ['user_id' => $user_id],  
    fn ($exists) => $exists ? [  
        'name' => $data['name'],  
        'email' => $data['email'],  
    ] : [  
        'name' => $data['name'],  
        'email' => $data['email'],  
        'marketable' => true,  
    ],  
) ;
```

## Обновление столбцов JSON

При обновлении столбца JSON вы должны использовать синтаксис `->` для обновления соответствующего ключа в объекте JSON. Эта операция поддерживается в MariaDB 10.3+, MySQL 5.7+ и PostgreSQL 9.5+:

```
$affected = DB::table('users')  
    ->where('id', 1)  
    ->update(['options->enabled' => true]);
```

## Увеличение и уменьшение отдельных значений

Конструктор запросов также содержит удобные методы увеличения или уменьшения значения конкретного столбца. Оба метода принимают по крайней мере один аргумент: столбец, который нужно изменить. Может быть указан второй аргумент, определяющий величину, на которую следует увеличить или уменьшить столбец:

```
DB::table('users')->increment('votes');  
  
DB::table('users')->increment('votes', 5);  
  
DB::table('users')->decrement('votes');  
  
DB::table('users')->decrement('votes', 5);
```

При необходимости вы также можете указать дополнительные столбцы для обновления во время операции увеличения или уменьшения:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Кроме того, вы можете одновременно увеличивать или уменьшать значения нескольких столбцов с помощью методов `incrementEach` и `decrementEach`:

```
DB::table('users')->incrementEach([
    'votes' => 5,
    'balance' => 100,
]);
```

## # Удаление

Метод `delete` может использоваться для удаления записей из таблицы. Он возвращает количество затронутых строк. Вы можете ограничить операторы `delete`, добавив метод `where` перед вызовом метода `delete`:

```
$deleted = DB::table('users')->delete();
$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

Если вы хотите очистить всю таблицу, что приведет к удалению всех записей из таблицы и сбросу автоинкрементного идентификатора на ноль, вы можете использовать метод `truncate`:

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

## Очистка таблицы и PostgreSQL

При очистке базы данных PostgreSQL будет применено поведение `CASCADE`. Это означает, что все связанные с внешним ключом записи в других таблицах также будут удалены.

## # Пессимистическая блокировка

Построитель запросов также включает несколько функций, которые помогут вам достичь «пессимистической блокировки» при выполнении ваших операторов `SELECT`. Чтобы выполнить оператор с «совместной блокировкой», вы можете вызвать метод `sharedLock` в запросе. Совместная блокировка предотвращает изменение выбранных строк до тех пор, пока ваша транзакция не будет зафиксирована:

```
DB::table('users')
    ->where('votes', '>', 100)
    ->sharedLock()
    ->get();
```

В качестве альтернативы вы можете использовать метод `lockForUpdate`. Блокировка «для обновления» предотвращает изменение выбранных записей или их выбор с помощью другой совместной блокировки:

```
DB::table('users')
    ->where('votes', '>', 100)
    ->lockForUpdate()
    ->get();
```

## # Отладка

Вы можете использовать методы `dd` или `dump` при построении запроса, чтобы отобразить связанные параметры запроса и сам SQL-запрос. Метод `dd` отобразит отладочную информацию и затем прекратит выполнение запроса. Метод `dump` отобразит информацию об отладке, но позволит продолжить выполнение запроса:

```
DB::table('users')->where('votes', '>', 100)->dd();
DB::table('users')->where('votes', '>', 100)->dump();
```

Методы `dumpRawSql` и `ddRawSql` могут быть вызваны для запроса, чтобы вывести SQL-запрос с правильно подставленными параметрами:

```
DB::table('users')->where('votes', '>', 100)->dumpRawSql();
```

```
DB::table('users')->where('votes', '>', 100)->ddRawSql();
```

# База данных · Постстраничная навигация

## # Введение

## # Основы использования

- # Разбиение результатов построителя запросов
- # Разбиение результатов Eloquent
- # Cursor-пагинация
- # Самостоятельное создание пагинатора
- # Настройка URL-адресов постраничной навигации

## # Отображение результатов постраничной навигации

- # Регулирование количества отображаемых ссылок
- # Преобразование результатов в JSON

## # Настройка вида пагинации

- # Использование Bootstrap

## # Методы экземпляра Paginator и LengthAwarePaginator

## # Методы экземпляра Cursor Paginator

## # Введение

В других фреймворках постраничная навигация может быть очень болезненной. Мы надеемся, что подход Laravel к разбиению на страницы станет глотком свежего воздуха. Пагинатор Laravel интегрирован с [построителем запросов](#) и [Eloquent ORM](#) и обеспечивает удобную, простую в использовании разбивку на страницы записей базы данных с нулевой конфигурацией.

По умолчанию HTML, генерируемый пагинатором, совместим с [фреймворком Tailwind CSS](#); однако, также доступна поддержка разбивки на страницы с использованием Bootstrap.

## Tailwind JIT

Если вы используете стандартные представления Laravel для разбивки на страницы Tailwind и механизм JIT Tailwind, вы должны убедиться, что ключ `content` файла

`tailwind.config.js` вашего приложения ссылаются на представления разбиения на страницы Laravel, чтобы их классы Tailwind не удалялись:

```
content: [
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
    './vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.blade.php'
],
```

## # Основы использования

### Разбиение результатов построителя запросов

Есть несколько способов разбить элементы на страницы. Самый простой – использовать метод `paginate` [построителя запросов](#) или в [запросе Eloquent](#). Метод `paginate` автоматически устанавливает «предел» и «смещение» в запросе на основе текущей страницы, просматриваемой пользователем. По умолчанию текущая страница определяется значением аргумента `page` строки HTTP-запроса. Это значение автоматически определяется Laravel, а также автоматически вставляется в ссылки, генерируемые пагинатором.

В этом примере единственный аргумент, переданный методу `paginate` – это количество элементов, которые вы хотите отображать «на каждой странице». В этом случае давайте укажем, что мы хотели бы отображать `15` элементов на странице:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать всех пользователей приложения.
     */
    public function index(): View
```

```
{  
    return view('user.index', [  
        'users' => DB::table('users')->paginate(15)  
    ]);  
}  
}
```

## Простая пагинация

Метод `paginate` подсчитывает общее количество записей, соответствующих запросу, перед извлечением записей из базы данных. Это сделано для того, чтобы пагинатор знал, сколько всего страниц с записями необходимо сформировать. Однако, если вы не планируете отображать общее количество страниц в пользовательском интерфейсе вашего приложения, запрос количества записей не нужен.

Следовательно, если вам нужно отображать только простые ссылки «Далее» и «Назад» в пользовательском интерфейсе вашего приложения, вы можете использовать метод `simplePaginate` для выполнения одного рационального запроса:

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

## Разбиение результатов Eloquent

Вы также можете разбивать запросы `Eloquent` на страницы. В этом примере мы разобьем модель `App\Models\User` на страницы и укажем, что мы планируем отображать 15 записей на странице. Как видите, синтаксис почти идентичен разбивке на страницы результатов построителя запросов:

```
use App\Models\User;  
  
$users = User::paginate(15);
```

Конечно, вы можете вызвать метод `paginate` после указания других ограничений для запроса, таких как выражения `where`:

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

Вы также можете использовать метод `simplePaginate` при разбиении на страницы моделей Eloquent:

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

Точно так же вы можете использовать метод `cursorPaginate` для курсорной пагинации моделей Eloquent:

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

## Несколько экземпляров пагинации на странице

Иногда вам может потребоваться отобразить два отдельных модуля пагинации на одном экране, который отображается вашим приложением. Однако, если оба экземпляра пагинации используют параметр строки запроса `page` для хранения текущей страницы, они будут конфликтовать. Чтобы разрешить этот конфликт, вы можете передать имя параметра строки запроса, который вы хотите использовать для хранения текущей страницы, через третий аргумент, предоставленный методам `paginate`, `simplePaginate` и `cursorPaginate`:

```
use App\Models\User;

$users = User::where('votes', '>', 100)->paginate(
    $perPage = 15, $columns = ['*'], $pageName = 'users'
);
```

## Cursor-пагинация

В то время как `paginate` и `simplePaginate` создают запросы с использованием SQL-оператора “`offset`”, Cursor-пагинация работает путем создания конструкции “`where`”, которая сравнивает значения упорядоченных столбцов, содержащихся в запросе, обеспечивая наиболее эффективную производительность базы данных среди всех возможных, доступную среди всех методов пагинации Laravel. Этот метод пагинации особенно хорошо подходит для больших наборов данных и пользовательских интерфейсов с “бесконечной” прокруткой.

В отличие от пагинации на основе смещения, которая включает номер страницы в строке запроса URL-адресов, сгенерированных средством постраничной

навигации, Cursor-пагинация помещает «cursor» в строку запроса. Курсор представляет собой закодированную строку, содержащую место, с которого следующий запрос с пагинацией должен начать постраничную навигацию, и направление, в котором он должен разбиваться на страницы:

```
http://localhost/users?cursor=eyJpZCI6MTUsI19wb2ludHNUb05leHRJdGVtcyI6dHJ1ZX0
```

Вы можете создать экземпляр Cursor-пагинации с помощью метода `cursorPaginate`, предлагаемого построителем запросов. Этот метод возвращает экземпляр `Illuminate\Pagination\CursorPaginator`:

```
$users = DB::table('users')->orderBy('id')->cursorPaginate(15);
```

После того как вы получили экземпляр Cursor-пагинации, вы можете [отобразить результаты постраничной навигации](#) как обычно при использовании методов `paginate` и `simplePaginate`. Для получения дополнительной информации о методах экземпляра, предлагаемых средством Cursor-пагинации, обратитесь к [документации по методам экземпляра Cursor Paginator](#).

Ваш запрос должен содержать “order by”, чтобы использовать Cursor-пагинацию. Кроме того, столбцы, по которым осуществляется сортировка запроса, должны принадлежать таблице, для которой вы используете пагинацию.

## “Cursor” против “Offset” пагинации

Чтобы проиллюстрировать различия между “Cursor” и “Offset” постраничной навигацией, давайте рассмотрим несколько примеров SQL-запросов. Оба следующих запроса будут отображать “вторую страницу” результатов для таблицы `users`, упорядоченных по `id`:

```
# Offset пагинация...
select * from users order by id asc limit 15 offset 15;
```

```
# Cursor пагинация...
select * from users where id > 15 order by id asc limit 15;
```

Cursor-пагинация предлагает следующие преимущества перед Offset-пагинацией:

- Для больших наборов данных Cursor-пагинация обеспечивать лучшую производительность, если столбцы "order by" проиндексированы. Это связано с тем, что предложение "offset" сканирует все ранее сопоставленные данные.
- Для наборов данных с частыми записями Offset-пагинация может пропускать записи или отображать дубликаты, если результаты были недавно добавлены или удалены со страницы, которую пользователь просматривает в данный момент.

Однако, Cursor-пагинация имеет следующие ограничения:

- Как и [simplePaginate](#), Cursor-пагинация может использоваться только для отображения ссылок "Далее" и "Назад" и не поддерживает создание ссылок с номерами страниц.
- Требуется, чтобы порядок был основан как минимум на одном уникальном столбце или на комбинации уникальных столбцов. Столбцы с `null` – значениями не поддерживаются.
- Выражения запросов с "order by" поддерживаются только в том случае, если они имеют псевдоним и также добавлены в "select".
- Выражения запросов с параметрами не поддерживаются.

## Самостоятельное создание пагинатора

По желанию можно вручную создать экземпляр пагинатора, передав ему массив элементов, которые у вас уже есть в памяти. Вы можете сделать это, создав экземпляр [Illuminate\Pagination\Paginator](#), [Illuminate\Pagination\LengthAwarePaginator](#) или [Illuminate\Pagination\CursorPaginator](#), в зависимости от ваших потребностей.

Классам [Paginator](#) и [CursorPaginator](#) не требуется знать общее количество элементов в результирующем наборе; однако, из-за этого у классов нет методов для получения индекса последней страницы. Класс [LengthAwarePaginator](#) принимает почти те же аргументы, что и [Paginator](#); однако, для этого требуется подсчет общего количества элементов в результирующем наборе.

Другими словами, `Paginator` соответствует методу `simplePaginate` построителя запросов, `CursorPaginator` соответствует методу `cursorPaginate`, а `LengthAwarePaginator` соответствует методу `paginate`.

При ручном создании экземпляра пагинатора вы должны самостоятельно «разрезать» массив результатов, который вы передаете в пагинатор. Если вы не знаете, как это сделать, ознакомьтесь с функцией PHP `array_slice`.

## Настройка URL-адресов постраничной навигации

По умолчанию ссылки, созданные пагинатором, будут соответствовать URI текущего запроса. Однако метод `withPath` пагинатора позволяет вам скорректировать URI, используемый пагинатором при генерации ссылок. Например, если вы хотите, чтобы пагинатор генерировал ссылки типа `http://example.com/admin/users?page=N`, вы должны передать `/admin/users` `withPath`:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->withPath('/admin/users');

    // ...
});
```

## Добавление значений в строку запроса

Вы можете добавить параметр в строку запроса навигационных ссылок с помощью метода `appends`. Например, чтобы добавить `sort=votes` к каждой ссылке пагинации, вы должны сделать следующий вызов `appends`:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);
```

```
$users->appends(['sort' => 'votes']);  
// ...  
});
```

Вы можете использовать метод `withQueryString`, если хотите добавить все значения строки текущего запроса к ссылкам постраничной навигации:

```
$users = User::paginate(15)->withQueryString();
```

## Добавление фрагментов хеша

Если вам нужно добавить «хеш-фрагмент» к URL-адресам, сгенерированным пагинатором, вы можете использовать метод `fragment`. Например, чтобы добавить `#users` в конец каждой навигационной ссылки, вы должны вызвать метод `fragment` следующим образом:

```
$users = User::paginate(15)->fragment('users');
```

## # Отображение результатов постраничной навигации

При вызове метода `paginate` вы получите экземпляр `Illuminate\Pagination\LengthAwarePaginator`, вызов метода `simplePaginate` возвращает экземпляр `Illuminate\Pagination\Paginator`. И, наконец, вызов метода `cursorPaginate` возвращает экземпляр `Illuminate\Pagination\CursorPaginator`.

Эти объекты содержат несколько методов, описывающих результирующий набор. В дополнение к этим вспомогательным методам, экземпляры пагинатора являются итераторами и могут быть перебраны как массив. Итак, как только вы получили результаты, вы можете отобразить результаты и отрисовать ссылки на страницы, используя `Blade`:

```
<div class="container">  
    @foreach ($users as $user)  
        {{ $user->name }}  
    @endforeach
```

```
</div>  
  
{{ $users->links() }}
```

Метод `links` отрисует ссылки на остальные страницы в результирующем наборе. Каждая из этих ссылок уже будет содержать соответствующую строковую переменную запроса `page`. Помните, что HTML, сгенерированный методом `links`, совместим с [фреймворком Tailwind CSS](#).

## Регулирование количества отображаемых ссылок

Когда пагинатор отображает навигационные ссылки, включающие номер текущей страницы, а также ссылки для трех страниц до и после текущей. Используя метод `onEachSide`, вы можете контролировать, сколько дополнительных ссылок отображается с каждой стороны от текущей страницы в среднем скользящем окне ссылок, созданного пагинатором:

```
{{ $users->onEachSide(5)->links() }}
```

## Преобразование результатов в JSON

Классы пагинатора Laravel реализуют контракт интерфейса `Illuminate\Contracts\Support\Jsonable` и содержат метод `toJson`, поэтому очень легко преобразовать результаты в JSON. Вы также можете преобразовать экземпляр пагинатора в JSON, вернув его из маршрута или действия контроллера:

```
use App\Models\User;  
  
Route::get('/users', function () {  
    return User::paginate();  
});
```

JSON из пагинатора будет включать метаинформацию, такую как `total`, `current_page`, `last_page` и другие. Записи результатов доступны через ключ `data` в массиве JSON. Вот пример JSON, созданного путем возврата экземпляра пагинатора из маршрута:

```
{  
    "total": 50,
```

```
"per_page": 15,  
"current_page": 1,  
"last_page": 4,  
"first_page_url": "http://laravel.app?page=1",  
"last_page_url": "http://laravel.app?page=4",  
"next_page_url": "http://laravel.app?page=2",  
"prev_page_url": null,  
"path": "http://laravel.app",  
"from": 1,  
"to": 15,  
"data": [  
    {  
        // Запись ...  
    },  
    {  
        // Запись ...  
    }  
]
```

## # Настройка вида пагинации

По умолчанию сгенерированные шаблоны для отображения навигационных ссылок, совместимы со структурой [фреймворка Tailwind CSS](#). Однако, если вы не используете Tailwind, вы можете определять свои собственные шаблоны для отображения этих ссылок. При вызове метода `links` в экземпляре пагинатора вы можете передать имя шаблона в качестве первого аргумента метода:

```
{{ $paginator->links('view.name') }}  
  
// Передача дополнительных данных в шаблон ...  
{ { $paginator->links('view.name', ['foo' => 'bar']) } }
```

Однако, самый простой способ отредактировать шаблоны постраничной навигации – это экспортировать их в каталог `resources/views/vendor` с помощью команды `vendor:publish`:

```
php artisan vendor:publish --tag=laravel-pagination
```

Эта команда поместит шаблоны в каталог `resources/views/vendor/pagination` вашего приложения. Файл `tailwind.blade.php` в этом каталоге соответствует шаблону постраничной навигации по умолчанию. Вы можете отредактировать этот файл для изменения HTML-кода навигации.

Если вы хотите назначить другой файл в качестве шаблона постраничной навигации по умолчанию, вы можете вызвать методы `defaultView` и `defaultSimpleView` пагинатора в методе `boot` вашего класса `App\Providers\AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        Paginator::defaultView('view-name');

        Paginator::defaultSimpleView('view-name');
    }
}
```

## Использование Bootstrap

Laravel содержит шаблоны постраничной навигации, созданные с использованием [Bootstrap CSS](#). Чтобы использовать эти шаблоны вместо шаблонов Tailwind по умолчанию, вы можете вызвать метод пагинатора `useBootstrapFour` или `useBootstrapFive` в методе `boot` класса `App\Providers\AppServiceProvider`:

```
use Illuminate\Pagination\Paginator;

/**
 * Загрузка любых служб приложения.
 */
public function boot(): void
{
```

```
Paginator::useBootstrapFive();
Paginator::useBootstrapFour();
}
```

## # Методы экземпляра Paginator и LengthAwarePaginator

Каждый экземпляр пагинатора содержит дополнительную информацию о постраничной навигации с помощью следующих методов:

| Метод  | Описание   |
|--|--|
| <code>\$paginator-&gt;count()</code>                     | Получить количество элементов для текущей страницы.  |
| <code>\$paginator-&gt;currentPage()</code>               | Получить номер текущей страницы.   |
| <code>\$paginator-&gt;firstItem()</code>                 | Получить номер первого элемента в результатах.   |
| <code>\$paginator-&gt;getOptions()</code>                | Получить параметры пагинатора.   |
| <code>\$paginator-&gt;getUrlRange(\$start, \$end)</code> | Создать диапазон URL-адресов для пагинации.  |
| <code>\$paginator-&gt;hasPages()</code>                  | Определить, достаточно ли элементов для разделения на несколько страниц.                                     |
| <code>\$paginator-&gt;hasMorePages()</code>              | Определить, есть ли еще элементы в хранилище данных.   |
| <code>\$paginator-&gt;items()</code>                     | Получить элементы для текущей страницы.  |
| <code>\$paginator-&gt;lastItem()</code>                  | Получить номер последнего элемента в результатах.  |
| <code>\$paginator-&gt;lastPage()</code>                  | Получить номер последней доступной страницы.<br>(Недоступно при использовании <code>simplePaginate</code> ). |
| <code>\$paginator-&gt;nextPageUrl()</code>               | Получить URL-адрес следующей страницы.   |

| <b>Метод</b>                                     | <b>Описание</b>  |
|--|--|
| <code>\$paginator-&gt;onFirstPage()</code>       | Определить, находится ли пагинатор на первой странице.   |
| <code>\$paginator-&gt;perPage()</code>           | Количество элементов, отображаемых на каждой странице.   |
| <code>\$paginator-&gt;previousPageUrl()</code>   | Получить URL-адрес предыдущей страницы.  |
| <code>\$paginator-&gt;total()</code>             | Определить общее количество элементов запроса в хранилище данных. (Недоступно при использовании <code>simplePaginate</code> ). |
| <code>\$paginator-&gt;url(\$page)</code>         | Получить URL-адрес для конкретного номера страницы.  |
| <code>\$paginator-&gt;getPageName()</code>       | Получить переменную строки запроса, используемую для хранения страницы.  |
| <code>\$paginator-&gt;setPageName(\$name)</code> | Установить переменную строки запроса, используемую для хранения страницы.  |
| <code>\$paginator-&gt;through(\$callback)</code> | Преобразуйте каждый элемент с использованием обратного вызова (замыкания).   |

## # Методы экземпляра Cursor Paginator

Каждый экземпляр Cursor-пагинатора предоставляет дополнительную информацию о постраничной навигации с помощью следующих методов:

| <b>Method</b>                         | <b>Description</b>                                  |
|---------------------------------------|---|
| <code>\$paginator-&gt;count()</code>  | Получить количество элементов для текущей страницы. |
| <code>\$paginator-&gt;cursor()</code> | Получить текущий экземпляр курсора.                 |

| <b>Method</b>                                  | <b>Description</b>   |
|--|--|
| <code>\$paginator-&gt;getOptions()</code>      | Получить параметры пагинатора.   |
| <code>\$paginator-&gt;hasPages()</code>        | Определить, достаточно ли элементов для разделения на несколько страниц. |
| <code>\$paginator-&gt;hasMorePages()</code>    | Определить, есть ли еще элементы в хранилище данных.                     |
| <code>\$paginator-&gt;getCursorName()</code>   | Получить переменную строки запроса, используемую для хранения курсора.   |
| <code>\$paginator-&gt;items()</code>           | Получить элементы для текущей страницы.                                  |
| <code>\$paginator-&gt;nextCursor()</code>      | Получить экземпляр курсора для следующего набора элементов.              |
| <code>\$paginator-&gt;nextPageUrl()</code>     | Получить URL-адрес следующей страницы.                                   |
| <code>\$paginator-&gt;onFirstPage()</code>     | Определить, находится ли пагинатор на первой странице.                   |
| <code>\$paginator-&gt;onLastPage()</code>      | Определить, находится ли пагинатор на последней странице.                |
| <code>\$paginator-&gt;perPage()</code>         | Количество элементов, отображаемых на каждой странице.                   |
| <code>\$paginator-&gt;previousCursor()</code>  | Получите экземпляр курсора для предыдущего набора элементов.             |
| <code>\$paginator-&gt;previousPageUrl()</code> | Получить URL-адрес предыдущей страницы.                                  |
| <code>\$paginator-&gt;setCursorName()</code>   | Установить переменную строки запроса, используемую для хранения курсора. |

| <b>Method</b>              | <b>Description</b>                                 |
|----------------------------|--|
| \$paginator->url(\$cursor) | Получить URL-адрес для данного экземпляра курсора. |

# База данных · Миграции

## # Введение

## # Генерация миграций

# Сжатие миграций

## # Структура миграций

## # Запуск миграций

# Откат миграций

## # Таблицы

# Создание таблиц

# Обновление таблиц

# Переименование / удаление таблиц

## # Столбцы

# Создание столбцов

# Доступные типы столбцов

# Модификаторы столбца

# Изменение столбцов

# Переименование столбцов

# Удаление столбцов

## # Индексы

# Создание индексов

# Переименование индексов

# Удаление индексов

# Ограничения внешнего ключа

## # События

## # Введение

Миграции похожи на контроль версий для вашей базы данных, позволяют вашей команде определять схемы базы данных приложения и совместно использовать их определение. Если вам когда-либо приходилось указывать товарищу по команде

вручную добавить столбец в его схему локальной базы данных после применения изменений в системе управления версиями, то вы столкнулись с проблемой, которую решает миграция базы данных.

[Фасад Schema](#) обеспечивает независимую от базы данных поддержку для создания и управления таблицами во всех поддерживаемых Laravel системах баз данных. В обычной ситуации, этот фасад используется для создания и изменения таблиц / столбцов базы данных во время миграции.

## # Генерация миграций

Чтобы сгенерировать новую миграцию базы данных, используйте команду `make:migration Artisan`. Эта команда поместит новый класс миграции в каталог `database/migrations` вашего приложения. Каждое имя файла миграции содержит временную метку, которая позволяет Laravel определять порядок применения миграций:

```
php artisan make:migration create_flights_table
```

Laravel будет использовать имя миграции, чтобы попытаться угадать имя таблицы и будет ли миграция создавать новую таблицу. Если Laravel может определить имя таблицы по имени миграции, то сгенерированный файл миграции будет предварительно заполнен указанной таблицей. В противном случае вы можете просто вручную указать таблицу в файле миграции.

Если вы хотите указать собственный путь для сгенерированной миграции, вы можете использовать параметр `--path` при выполнении команды `make:migration`. Указанный путь должен быть относительно базового пути вашего приложения.

Заготовки (stubs) миграции можно настроить с помощью [публикации заготовок](#).

## Сжатие миграций

По мере создания приложения вы можете со временем накапливать все больше и больше миграций. Это может привести к тому, что ваш каталог `database/migrations` станет раздутым из-за потенциально сотен миграций. Если хотите, то можете «сжать» свои миграции в один файл SQL. Для начала выполните команду `schema:dump`:

```
php artisan schema:dump  
  
# Выгрузить текущую схему БД и удалить все существующие миграции ...  
php artisan schema:dump --prune
```

В результате выполнения этой команды Laravel запишет дамп базы данных в каталог `database/schema` вашего приложения. Теперь, при запуске миграции базы данных, Laravel сначала выполнит SQL-операторы дампа (если никакие другие миграции не выполнялись). Затем Laravel выполнит все оставшиеся миграции, которые не были включены в дамп схемы БД.

Если ваши тесты приложения используют другое подключение к базе данных, чем то, которое вы обычно используете во время локальной разработки, убедитесь, что вы создали файл схемы с использованием этого подключения к базе данных, чтобы ваши тесты могли создать базу данных. Вы можете сделать это после создания файла схемы для базы данных, которую обычно используете во время локальной разработки:

```
php artisan schema:dump  
php artisan schema:dump --database=testing --prune
```

Вы должны передать файл схемы базы данных в систему управления версиями, чтобы другие разработчики вашей команды могли быстро воссоздать исходную структуру базы данных вашего приложения.

Сжатие миграции доступно только для баз данных MariaDB, MySQL, PostgreSQL и SQLite и использует клиент командной строки базы данных.

# # Структура миграций

Класс миграции содержит два метода: `up` и `down`. Метод `up` используется для добавления новых таблиц, столбцов или индексов в вашу базу данных, тогда как метод `down` должен отменять операции, выполняемые методом `up`.

В обоих этих методах вы можете использовать построитель схем Laravel для выразительного создания и изменения таблиц. Чтобы узнать обо всех методах, доступных построителю `Schema`, [просмотрите его документацию](#). Например, следующая миграция создает таблицу `flights`:

```
<?php

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

return new class extends Migration
{
    /**
     * Запустить миграцию.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Обратить миграции.
     */
    public function down(): void
    {
        Schema::drop('flights');
    }
}
```

## Указание соединения миграции

Если ваша миграция будет использовать соединение с базой данных, отличное от соединения с базой данных по умолчанию, то необходимо установить свойство `$connection` миграции:

```
/**
 * Соединение с БД, которое должно использоваться миграцией.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Запустить миграцию.
 */
public function up(): void
{
    // ...
}
```

## # Запуск миграций

Чтобы запустить все незавершенные миграции, выполните команду `migrate` Artisan:

```
php artisan migrate
```

Если вы хотите узнать, какие миграции уже выполнены, то вы можете использовать команду `migrate:status` Artisan:

```
php artisan migrate:status
```

Если вы хотите посмотреть SQL-запросы, которые будут выполнены миграциями, но при этом не запускать их фактически, вы можете добавить флаг `--pretend` к команде `migrate`:

```
php artisan migrate --pretend
```

## Изолированное выполнение миграций

Если вы развертываете свое приложение на нескольких серверах и выполняете миграции в рамках процесса развертывания, вероятно, вам не захочется, чтобы два сервера попытались выполнить миграцию базы данных одновременно. Для избежания этого вы можете использовать опцию `isolated` при вызове команды `migrate`.

Когда указана опция `isolated`, Laravel получит атомарный блокировщик с использованием драйвера кэша вашего приложения перед попыткой запуска миграций. Все другие попытки выполнить команду `migrate` в то время, как этот блокировщик удерживается, не будут выполнены; однако команда все равно завершит свое выполнение с успешным кодом статуса:

```
php artisan migrate --isolated
```

Для использования этой функции ваше приложение должно использовать драйвер кэша `memcached`, `redis`, `dynamodb`, `database`, `file` или `array` как драйвер кэша по умолчанию. Кроме того, все серверы должны общаться с одним и тем же центральным сервером кэша.

## Принудительный запуск миграции в рабочем окружении

Некоторые операции миграции являются деструктивными, что означает, что они могут привести к потере данных. Чтобы защитить вас от запуска этих команд для вашей производственной базы данных, от вас потребуется подтверждение перед выполнением команд. Чтобы команды запускались без подтверждения, используйте флаг `--force`:

```
php artisan migrate --force
```

## Откат миграций

Чтобы откатить последнюю операцию миграции, вы можете использовать команду `rollback` Artisan. Эта команда откатывает последний «пакет» миграций, который

может включать несколько файлов миграции:

```
php artisan migrate:rollback
```

Вы можете откатить ограниченное количество миграций, указав параметр `step` для команды `rollback`. Например, следующая команда откатит последние пять миграций:

```
php artisan migrate:rollback --step=5
```

Вы можете откатить определенную “партию” миграций, указав опцию `batch` команде `rollback`, где значение опции `batch` соответствует значению партии в таблице `migrations` вашего приложения. Например, следующая команда откатит все миграции в партии третьей:

```
php artisan migrate:rollback --batch=3
```

Если вы хотите посмотреть SQL-запросы, которые будут выполнены миграциями без их фактического выполнения, вы можете добавить флаг `--pretend` к команде `migrate:rollback`:

```
php artisan migrate:rollback --pretend
```

Команда `migrate:reset` откатит все миграции вашего приложения:

```
php artisan migrate:reset
```

## Откат и миграция с помощью одной команды

Команда `migrate:refresh` откатит все ваши миграции, а затем выполнит команду `migrate`. Эта команда эффективно воссоздает всю вашу базу данных:

```
php artisan migrate:refresh
```

```
// Обновляем базу данных и запускаем все наполнители базы данных ...
php artisan migrate:refresh --seed
```

Вы можете откатить и повторно запустить ограниченное количество миграций, указав параметр `step` для команды `refresh`. Например, следующая команда откатит и повторно запустит последние пять миграций:

```
php artisan migrate:refresh --step=5
```

## Удаление всех таблиц с последующей миграцией

Команда `migrate:fresh` удалит все таблицы из базы данных, а затем выполнит команду `migrate`:

```
php artisan migrate:fresh
```

```
php artisan migrate:fresh --seed
```

По умолчанию команда `migrate:fresh` удаляет только таблицы из соединения с базой данных по умолчанию. Однако вы можете использовать опцию `--database`, чтобы указать соединение с базой данных, которое следует использовать. Имя соединения с базой данных должно соответствовать имени, определенному в [файле конфигурации базы данных](#) вашего приложения:

```
php artisan migrate:fresh --database=admin
```

Команда `migrate:fresh` удалит все таблицы базы данных независимо от их префикса. Эту команду следует использовать с осторожностью при разработке в базе данных, которая используется совместно с другими приложениями.

## # Таблицы

# Создание таблиц

Чтобы создать новую таблицу базы данных, используйте метод `create` фасада `Schema`. Метод `create` принимает два аргумента: первый – это имя таблицы, а второй – замыкание, которое получает объект `Blueprint`, используемый для определения новой таблицы:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

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

При создании таблицы вы можете использовать любой из [методов столбцов](#) построителя схемы для определения столбцов таблицы.

## Определение наличия таблицы / столбца

Вы можете определить наличие таблицы, столбца или индекса с помощью методов `hasTable`, `hasColumn` и `hasIndex`, соответственно:

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

if (Schema::hasColumn('users', 'email')) {
    // Таблица `users` существует и содержит столбец `email` ...
}

if (Schema::hasIndex('users', ['email'], 'unique')) {
    // Таблица `users` существует и имеет уникальный индекс в столбце `email`...
}
```

## Соединение с базой данных и параметры таблицы

Если вы хотите выполнить операцию схемы с подключением, которое не является подключением к базе данных по умолчанию для вашего приложения, используйте метод `connection`:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

Кроме того, некоторые другие свойства и методы могут использоваться для определения других аспектов создания таблицы. Свойство `engine` используется для указания механизма хранения таблицы при использовании MariaDB или MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine('InnoDB');

    // ...
});
```

Свойства `charset` и `collation` могут использоваться для указания набора символов и сопоставления для создаваемой таблицы при использовании MariaDB или MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset('utf8mb4');
    $table->collation('utf8mb4_unicode_ci');

    // ...
});
```

Метод `temporary` используется, чтобы указать, что таблица должна быть «временной». Временные таблицы видны только текущему сеансу соединения базы данных и автоматически удаляются при закрытии соединения:

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

    // ...
});
```

Если вы хотите добавить “комментарий” к таблице базы данных, вы можете вызвать метод `comment` на экземпляре таблицы. Комментарии к таблицам поддерживаются только в MariaDB, MySQL и Postgres:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->comment('Business calculations');

    // ...
});
```

Этот код позволит вам добавить комментарий “Business calculations” к таблице “calculations” в вашей базе данных. Это может быть полезно для документации и описания цели таблицы.

## Обновление таблиц

Метод `table` фасада `Schema` используется для обновления существующих таблиц. Подобно методу `create`, метод `table` принимает два аргумента: имя таблицы и замыкание, которое получает экземпляр `Blueprint`, используемый для добавления столбцов или индексов в таблицу:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

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

## Переименование / удаление таблиц

Чтобы переименовать существующую таблицу базы данных, используйте метод `rename`:

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

Чтобы удалить существующую таблицу, вы можете использовать методы `drop` или `dropIfExists`:

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

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

## Переименование таблиц с внешними ключами

Перед переименованием таблицы вы должны убедиться, что любые ограничения внешнего ключа в таблице имеют явное имя в ваших файлах миграции, вместо того, чтобы позволять Laravel назначать имя на основе соглашения. В противном случае имя ограничения внешнего ключа будет ссылаться на имя старой таблицы.

## # Столбцы

### Создание столбцов

Метод `table` фасада `Schema` используется для обновления существующих таблиц. Как и метод `create`, метод `table` принимает два аргумента: имя таблицы и замыкание, которое получает экземпляр `Illuminate\Database\Schema\Blueprint`, используемый для добавления столбцов в таблицу:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

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

### Доступные типы столбцов

Построитель схем Blueprint предлагает множество методов, соответствующих различным типам столбцов, которые вы можете добавить в таблицы базы данных. Все доступные методы перечислены в таблице ниже:

[bigIncrements](#)

[bigInteger](#)

[binary](#)

[boolean](#)

[char](#)

[dateTimeTz](#)

[dateTime](#)

date  
decimal  
double  
enum  
float  
foreignId  
foreignIdFor  
foreignUlid  
foreignUuid  
geography  
geometry  
id  
increments  
integer  
ipAddress  
json  
jsonb  
longText  
macAddress  
mediumIncrements  
mediumInteger  
mediumText  
morphs  
nullableMorphs  
nullableTimestamps  
nullableUlidMorphs  
nullableUuidMorphs  
rememberToken  
set  
smallIncrements  
smallInteger  
softDeletesTz  
softDeletes  
string  
text  
timeTz

[time](#)  
[timestampTz](#)  
[timestamp](#)  
[timestampsTz](#)  
[timestamps](#)  
[tinyIncrements](#)  
[tinyInteger](#)  
[tinyText](#)  
[unsignedBigInteger](#)  
[unsignedInteger](#)  
[unsignedMediumInteger](#)  
[unsignedSmallInteger](#)  
[unsignedTinyInteger](#)  
[ulidMorphs](#)  
[uuidMorphs](#)  
[ulid](#)  
[uuid](#)  
[year](#)

## bigIncrements()

Метод `bigIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED BIGINT` (первичный ключ):

```
$table->bigIncrements('id');
```

## bigInteger()

Метод `bigInteger` создает эквивалент столбца `BIGINT`:

```
$table->bigInteger('votes');
```

## binary()

Метод `binary` создает эквивалент столбца `BLOB`:

```
$table->binary('photo');
```

При использовании MySQL, MariaDB или SQL Server вы можете передать аргументы `length` и `fixed` для создания эквивалентного столбца `VARBINARY` или `BINARY`:

```
$table->binary('data', length: 16); // VARBINARY(16)
```

```
$table->binary('data', length: 16, fixed: true); // BINARY(16)
```

## boolean()

Метод `boolean` создает эквивалент столбца `BOOLEAN`:

```
$table->boolean('confirmed');
```

## char()

Метод `char` создает эквивалент столбца `CHAR` указанной длины:

```
$table->char('name', length: 100);
```

## dateTimeTz()

Метод `dateTimeTz` создает эквивалент столбца `DATETIME` (с часовым поясом) с необязательной точностью до долей секунды:

```
$table->dateTimeTz('created_at', precision: 0);
```

## dateTime()

Метод `dateTime` создает эквивалент столбца `DATETIME` с необязательной точностью до долей секунды:

```
$table->dateTime('created_at', precision: 0);
```

## date()

Метод `date` создает эквивалент столбца `DATE`:

```
$table->date('created_at');
```

## decimal()

Метод `decimal` создает эквивалент столбца `DECIMAL` с точностью (общее количество цифр) и масштабом (десятичные цифры):

```
$table->decimal('amount', total: 8, places: 2);
```

## double()

Метод `double` создает эквивалент столбца `DOUBLE`:

```
$table->double('amount');
```

## enum()

Метод `enum` создает эквивалент столбца `ENUM` с указанием допустимых значений:

```
$table->enum('difficulty', ['easy', 'hard']);
```

## float()

Метод `float` создает эквивалент столбца `FLOAT` с заданной точностью:

```
$table->float('amount', precision: 53);
```

## foreignId()

Метод `foreignId` создает эквивалент столбца `UNSIGNED BIGINT`:

```
$table->foreignKey('user_id');
```

## foreignIdFor()

Метод `foreignKeyFor` добавляет столбец с именем `{column}_id`, эквивалентный для заданного класса модели. Тип столбца будет `UNSIGNED BIGINT`, `CHAR(36)` или `CHAR(26)`, в зависимости от типа ключа модели:

```
$table->foreignKeyFor(User::class);
```

## foreignUlid()

Метод `foreignKey` создает столбец, эквивалентный `ULID`:

```
$table->foreignKey('user_id');
```

## foreignUuid()

Метод `foreignKey` создает эквивалент столбца `UUID`:

```
$table->foreignKey('user_id');
```

## geography()

Метод `geography` создает эквивалент столбца `GEOGRAPHY` с заданным пространственным типом и SRID (идентификатором пространственной системы отсчета):

```
$table->geography('coordinates', subtype: 'point', srid: 4326);
```

Поддержка пространственных типов зависит от драйвера вашей базы данных. Пожалуйста, обратитесь к документации вашей базы данных. Если ваше приложение использует базу данных

PostgreSQL, вам необходимо установить расширение [PostGIS](#), прежде чем можно будет использовать метод [geography](#).

## geometry()

Метод [geometry](#) создает эквивалент столбца [GEOMETRY](#) с заданным пространственным типом и SRID (идентификатором пространственной системы отсчета):

```
$table->geometry('positions', subtype: 'point', srid: 0);
```

Поддержка пространственных типов зависит от драйвера вашей базы данных. Пожалуйста, обратитесь к документации вашей базы данных. Если ваше приложение использует базу данных PostgreSQL, вам необходимо установить расширение [PostGIS](#), прежде чем можно будет использовать метод [geometry](#).

## id()

Метод [id](#) является псевдонимом метода [bigIncrements](#). По умолчанию метод создает столбец [id](#); однако, вы можете передать имя столбца, если хотите присвоить столбцу другое имя:

```
$table->id();
```

## increments()

Метод [increments](#) создает эквивалент автоинкрементного столбца [UNSIGNED INTEGER](#) в качестве первичного ключа:

```
$table->increments('id');
```

## integer()

Метод `integer` создает эквивалент столбца `INTEGER`:

```
$table->integer('votes');
```

## ipAddress()

Метод `ipAddress` создает эквивалент столбца `VARCHAR`:

```
$table->ipAddress('visitor');
```

При использовании PostgreSQL будет создан столбец `INET`.

## json()

Метод `json` создает эквивалент столбца `JSON`:

```
$table->json('options');
```

## jsonb()

Метод `jsonb` создает эквивалент столбца `JSONB`:

```
$table->jsonb('options');
```

## longText()

Метод `longText` создает эквивалент столбца `LONGTEXT`:

```
$table->longText('description');
```

При использовании MySQL или MariaDB вы можете применить к столбцу `binary` набор символов, чтобы создать эквивалентный столбец `LONGBLOB`:

```
$table->longText('data')->charset('binary'); // LONGBLOB
```

## macAddress()

Метод `macAddress` создает столбец, предназначенный для хранения МАС-адреса. Некоторые системы баз данных, такие как PostgreSQL, имеют специальный тип столбца для этого типа данных. Другие системы баз данных будут использовать столбец строкового эквивалента:

```
$table->macAddress('device');
```

## mediumIncrements()

Метод `mediumIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED MEDIUMINT` в качестве первичного ключа:

```
$table->mediumIncrements('id');
```

## mediumInteger()

Метод `mediumInteger` создает эквивалент столбца `MEDIUMINT`:

```
$table->mediumInteger('votes');
```

## mediumText()

Метод `mediumText` создает эквивалент столбца `MEDIUMTEXT`:

```
$table->mediumText('description');
```

При использовании MySQL или MariaDB вы можете применить к столбцу `binary` набор символов, чтобы создать эквивалентный столбец `MEDIUMBLOB`:

```
$table->mediumText('data')->charset('binary'); // MEDIUMBLOB
```

## morphs()

Метод `morphs` – это удобный метод, который добавляет эквивалент столбца `{column}_id` и столбца `{column}_type` с типом данных `VARCHAR`. Тип данных столбца `{column}_id` будет `UNSIGNED BIGINT, CHAR(36)` или `CHAR(26)`, в зависимости от типа ключа модели.

Этот метод предназначен для использования при определении столбцов, необходимых для полиморфного [отношения Eloquent](#). В следующем примере будут созданы столбцы `taggable_id` и `taggable_type`:

```
$table->morphs('taggable');
```

## nullableTimestamps()

Метод `nullableTimestamps` является псевдонимом метода [`timestamps`](#):

```
$table->nullableTimestamps(precision: 0);
```

## nullableMorphs()

Метод аналогичен методу `morphs`; тем не менее, создаваемый столбец будет иметь значение `NULL`:

```
$table->nullableMorphs('taggable');
```

## nullableUlidMorphs()

Этот метод аналогичен методу `ulidMorphs`; однако создаваемые столбцы будут “`nullable`” (допускающими значение `null`):

```
$table->nullableUlidMorphs('taggable');
```

## nullableUuidMorphs()

Метод аналогичен методу `uuidMorphs`; тем не менее, создаваемый столбец будет иметь значение `NULL`:

```
$table->nullableUuidMorphs('taggable');
```

## rememberToken()

Метод `rememberToken` создает NULL-эквивалент столбца `VARCHAR(100)`, предназначенный для хранения текущего [токена аутентификации](#):

```
$table->rememberToken();
```

## set()

Метод `set` создает эквивалент столбца `SET` с заданным списком допустимых значений:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

## smallIncrements()

Метод `smallIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED SMALLINT` в качестве первичного ключа:

```
$table->smallIncrements('id');
```

## smallInteger()

Метод `smallInteger` создает эквивалент столбца `SMALLINT`:

```
$table->smallInteger('votes');
```

## softDeletesTz()

Метод `softDeletesTz` добавляет NULL-эквивалент столбца `TIMESTAMP` (с часовым поясом) с необязательной точностью до долей секунды. Этот столбец предназначен для хранения временной метки `deleted_at`, необходимой для функции «программного удаления» Eloquent:

```
$table->softDeletesTz('deleted_at', precision: 0);
```

## softDeletes()

Метод `softDeletes` добавляет NULL-эквивалент столбца `TIMESTAMP` с необязательной точностью до долей секунды. Этот столбец предназначен для хранения временной метки `deleted_at`, необходимой для функции «программного удаления» Eloquent:

```
$table->softDeletes('deleted_at', precision: 0);
```

## string()

Метод `string` создает эквивалент столбца `VARCHAR` указанной длины:

```
$table->string('name', length: 100);
```

## text()

Метод `text` создает эквивалент столбца `TEXT`:

```
$table->text('description');
```

При использовании MySQL или MariaDB вы можете применить к столбцу `binary` набор символов, чтобы создать эквивалент столбца `BLOB`:

```
$table->text('data')->charset('binary'); // BLOB
```

## timeTz()

Метод `timeTz` создает эквивалент столбца `TIME` (с часовым поясом) с необязательной точностью до долей секунды:

```
$table->timeTz('sunrise', precision: 0);
```

## time()

Метод `time` создает эквивалент столбца `TIME` с необязательной точностью до долей секунды:

```
$table->time('sunrise', precision: 0);
```

## timestampTz()

Метод `timestampTz` создает эквивалент столбца `TIMESTAMP` (с часовым поясом) с необязательной точностью до долей секунды:

```
$table->timestampTz('added_at', precision: 0);
```

## timestamp()

Метод `timestamp` создает эквивалент столбца `TIMESTAMP` с необязательной точностью до долей секунды:

```
$table->timestamp('added_at', precision: 0);
```

## timestampsTz()

Метод `timestampsTz` создает столбцы `created_at` и `updated_at`, эквивалентные `TIMESTAMP` (с часовым поясом) с необязательной точностью до долей секунды:

```
$table->timestampsTz(precision: 0);
```

## timestamps()

Метод `timestamps` method creates `created_at` and `updated_at` `TIMESTAMP` с необязательной точностью до долей секунды:

```
$table->timestamps(precision: 0);
```

## **tinyIncrements()**

Метод `tinyIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED TINYINT` в качестве первичного ключа:

```
$table->tinyIncrements('id');
```

## **tinyInteger()**

Метод `tinyInteger` создает эквивалент столбца `TINYINT`:

```
$table->tinyInteger('votes');
```

## **tinyText()**

Метод `tinyText` создаёт эквивалент столбца `TINYTEXT`:

```
$table->tinyText('notes');
```

When utilizing MySQL or MariaDB, you may apply a `binary` character set to the column in order to create a `TINYBLOB` equivalent column: При использовании MySQL или MariaDB вы можете применить к столбцу `binary` набор символов, чтобы создать эквивалентный столбец `TINYBLOB`:

```
$table->tinyText('data')->charset('binary'); // TINYBLOB
```

## **unsignedBigInteger()**

Метод `unsignedBigInteger` создает эквивалент столбца `UNSIGNED BIGINT`:

```
$table->unsignedBigInteger('votes');
```

## **unsignedInteger()**

Метод `unsignedInteger` создает эквивалент столбца `UNSIGNED INTEGER`:

```
$table->unsignedInteger('votes');
```

## unsignedMediumInteger()

Метод `unsignedMediumInteger` создает эквивалент столбца `UNSIGNED MEDIUMINT`:

```
$table->unsignedMediumInteger('votes');
```

## unsignedSmallInteger()

Метод `unsignedSmallInteger` создает эквивалент столбца `UNSIGNED SMALLINT`:

```
$table->unsignedSmallInteger('votes');
```

## unsignedTinyInteger()

Метод `unsignedTinyInteger` создает эквивалент столбца `UNSIGNED TINYINT`:

```
$table->unsignedTinyInteger('votes');
```

## Метод ulidMorphs()

Метод `ulidMorphs` – это удобный метод, который добавляет эквивалент столбца `{column}_id` типа `CHAR(26)` и столбца `{column}_type` типа `VARCHAR`.

Этот метод предназначен для использования при определении столбцов, необходимых для полиморфных [Eloquent отношений](#), которые используют ULID идентификаторы. В следующем примере будут созданы столбцы `taggable_id` и `taggable_type`:

```
$table->ulidMorphs('taggable');
```

## uuidMorphs()

Метод `uuidMorphs` – это удобный метод, который добавляет эквивалент столбца `CHAR(36) ({column}_id)` и эквивалент столбца `VARCHAR ({column}_type)`.

Этот метод предназначен для использования при определении столбцов, необходимых для полиморфного [отношения Eloquent](#), использующего идентификаторы UUID. В следующем примере будут созданы столбцы `taggable_id` и `taggable_type`:

```
$table->uuidMorphs('taggable');
```

## Метод ulid()

Метод `ulid` создает столбец, эквивалентный [ULID](#):

```
$table->ulid('id');
```

Метод `ulid` создает столбец, эквивалентный [ULID](#) и присваивает ему имя 'id'.

## uuid()

Метод `uuid` создает эквивалент столбца [UUID](#):

```
$table->uuid('id');
```

## year()

Метод `year` создает эквивалент столбца [YEAR](#):

```
$table->year('birth_year');
```

# Модификаторы столбца

В дополнение к типам столбцов, перечисленным выше, есть несколько «модификаторов» столбцов, которые вы можете использовать при добавлении столбца в таблицу базы данных. Например, чтобы сделать столбец «допускающим значение NULL», вы можете использовать метод `nullable`:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
```

```

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

```

В следующей таблице представлены все доступные модификаторы столбцов. В этот список не входят [модификаторы индексов](#):

| Модификатор                                       | Описание   |
|---|--|
| <code>-&gt;after('column')</code>                 | Поместить столбец «после» другого столбца (MariaDB / MySQL).                         |
| <code>-&gt;autoIncrement()</code>                 | Установить столбцы INTEGER как автоинкрементные (первичный ключ).                    |
| <code>-&gt;charset('utf8mb4')</code>              | Указать набор символов для столбца (MariaDB / MySQL).                                |
| <code>-&gt;collation('utf8mb4_unicode_ci')</code> | Укажите параметры сортировки для столбца.  |
| <code>-&gt;comment('my comment')</code>           | Добавить комментарий к столбцу (MariaDB / MySQL / PostgreSQL).                       |
| <code>-&gt;default(\$value)</code>                | Указать значение «по умолчанию» для столбца.   |
| <code>-&gt;first()</code>                         | Поместить столбец «первым» в таблице (MariaDB / MySQL).                              |
| <code>-&gt;from(\$integer)</code>                 | Установить начальное значение автоинкрементного поля (MariaDB / MySQL / PostgreSQL). |
| <code>-&gt;invisible()</code>                     | Сделать столбец "невидимым" для запросов <code>SELECT *</code> (MariaDB / MySQL).    |
| <code>-&gt;nullable(\$value = true)</code>        | Позволить (по умолчанию) значения <code>NULL</code> для вставки в столбец.           |

| <b>Модификатор</b>                          | <b>Описание</b>   |
|---|---|
| <code>-&gt;storedAs(\$expression)</code>    | Создать сохраненный генерируемый столбец (MariaDB / MySQL / PostgreSQL / SQLite).   |
| <code>-&gt;unsigned()</code>                | Установить столбцы <code>INTEGER</code> как <code>UNSIGNED</code> (MariaDB / MySQL).  |
| <code>-&gt;useCurrent()</code>              | Установить столбцы <code>TIMESTAMP</code> для использования <code>CURRENT_TIMESTAMP</code> в качестве значения по умолчанию.        |
| <code>-&gt;useCurrentOnUpdate()</code>      | Установить столбцы <code>TIMESTAMP</code> для использования <code>CURRENT_TIMESTAMP</code> при обновлении записи (MariaDB / MySQL). |
| <code>-&gt;virtualAs(\$expression)</code>   | Создать виртуальный генерируемый столбец (MariaDB / MySQL / SQLite).  |
| <code>-&gt;generatedAs(\$expression)</code> | Создать столбец идентификаторов с указанными параметрами последовательности (PostgreSQL).   |
| <code>-&gt;always()</code>                  | Определить приоритет значений последовательности над вводом для столбца идентификаторов (PostgreSQL).                               |

## Выражения для значений по умолчанию

Модификатор `default` принимает значение или экземпляр `Illuminate\Database\Query\Expression`. Использование экземпляра `Expression` не позволит Laravel заключить значение в кавычки и позволит вам использовать функции, специфичные для базы данных. Одна из ситуаций, когда это особенно полезно, когда вам нужно назначить значения по умолчанию для столбцов JSON:

```
<?php
```

```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
```

```
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    /**
     * Запустить миграцию.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('JSON_ARRAY()'));
            $table->timestamps();
        });
    }
}
```

Поддержка выражений по умолчанию зависит от вашего драйвера базы данных, версии базы данных и типа поля. См. документацию к вашей базе данных.

## Порядок столбцов

Метод `after` добавляет набор столбцов после указанного существующего столбца в схеме базы данных MariaDB или MySQL:

```
$table->after('password', function (Blueprint $table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

## Изменение столбцов

Метод `change` позволяет вам изменять тип и атрибуты существующих колонок. Например, вы можете захотеть увеличить размер колонки типа `string`. Чтобы увидеть метод `change` в действии, давайте увеличим размер колонки `name` с 25 до 50.

Для этого мы просто определяем новое состояние колонки и затем вызываем метод `change`:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

При изменении колонки вы должны явно включить все модификаторы, которые вы хотите сохранить в определении колонки – любой пропущенный атрибут будет удален. Например, чтобы сохранить атрибуты `unsigned`, `default` и `comment`, вам нужно вызывать каждый модификатор явно при изменении колонки:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')->unsigned()->default(1)->comment('мой комментарий')->change();
});
```

Метод `change` не меняет индексы столбца. Поэтому вы можете использовать модификаторы индекса, чтобы явно добавлять или удалять индекс при изменении столбца:

```
// Add an index...
$table->bigIncrements('id')->primary()->change();

// Drop an index...
$table->char('postal_code', 10)->unique(false)->change();
```

## Переименование столбцов

Для переименования столбца вы можете использовать метод `renameColumn`, предоставленный строителем схемы:

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

## Удаление столбцов

Для удаления столбца вы можете использовать метод `dropColumn` в билдере схемы:

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

Вы можете удалить несколько столбцов из таблицы, передав массив имен столбцов методу `dropColumn`:

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

## Доступные псевдонимы команд

Laravel содержит несколько удобных методов, связанных с удалением общих типов столбцов. Каждый из этих методов описан в таблице ниже:

| Команда   | Описание  |
|---|---|
| <code>\$table-&gt;dropMorphs('morphable');</code> | Удалить столбцы <code>morphable_id</code> и <code>morphable_type</code> . |
| <code>\$table-&gt;dropRememberToken();</code>     | Удалить столбец <code>remember_token</code> .                             |
| <code>\$table-&gt;dropSoftDeletes();</code>       | Удалить столбец <code>deleted_at</code> .                                 |
| <code>\$table-&gt;dropSoftDeletesTz();</code>     | Псевдоним <code>dropSoftDeletes()</code> .                                |
| <code>\$table-&gt;dropTimestamps();</code>        | Удалить столбцы <code>created_at</code> и <code>updated_at</code> .       |
| <code>\$table-&gt;dropTimestampsTz();</code>      | Псевдоним <code>dropTimestamps()</code> .                                 |

## # Индексы

### Создание индексов

Постройтель схем Laravel поддерживает несколько типов индексов. В следующем примере создается новый столбец `email` и указывается, что его значения должны

быть уникальными. Чтобы создать индекс, мы можем связать метод `unique` с определением столбца:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

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

В качестве альтернативы вы можете создать индекс после определения столбца. Для этого вы должны вызвать метод `unique` построителя схемы Blueprint. Этот метод принимает имя столбца, который должен получить уникальный индекс:

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

Вы даже можете передать массив столбцов методу индекса для создания составного индекса:

```
$table->index(['account_id', 'created_at']);
```

При создании индекса Laravel автоматически сгенерирует имя индекса на основе таблицы, имен столбцов и типа индекса, но вы можете передать второй аргумент методу, чтобы указать имя индекса самостоятельно:

```
$table->unique('email', 'unique_email');
```

## Доступные типы индексов

Построитель схем Laravel содержит методы для создания каждого типа индекса, поддерживаемого Laravel. Каждый метод индекса принимает необязательный второй аргумент для указания имени индекса. Если не указано, то имя будет производным от имен таблицы и столбцов, используемых для индекса, а также типа индекса. Все доступные методы индекса описаны в таблице ниже:

| Команда  | Описание   |
|--|--|
| <code>\$table-&gt;primary('id');</code>                            | Добавить первичный ключ.   |
| <code>\$table-&gt;primary(['id', 'parent_id']);</code>             | Добавить составной ключ.   |
| <code>\$table-&gt;unique('email');</code>                          | Добавить уникальный индекс.  |
| <code>\$table-&gt;index('state');</code>                           | Добавляет простой индекс.  |
| <code>\$table-&gt;fulltext('body');</code>                         | Добавляет полнотекстовый индекс (MariaDB / MySQL / PostgreSQL).    |
| <code>\$table-&gt;fulltext('body')-&gt;language('english');</code> | Добавляет полнотекстовый индекс для указанного языка (PostgreSQL). |
| <code>\$table-&gt;spatialIndex('location');</code>                 | Добавляет пространственный индекс (кроме SQLite).                  |

## Переименование индексов

Чтобы переименовать индекс, вы можете использовать метод `renameIndex` построителя схемы Blueprint. Этот метод принимает текущее имя индекса в качестве первого аргумента и желаемое имя в качестве второго аргумента:

```
$table->renameIndex('from', 'to')
```

## Удаление индексов

Чтобы удалить индекс, вы должны указать имя индекса. По умолчанию Laravel автоматически назначает имя индекса на основе имени таблицы, имени индексированного столбца и типа индекса. Вот некоторые примеры:

| Команда   | Описание  |
|---|---|
| <code>\$table-&gt;dropPrimary('users_id_primary');</code>               | Удалить первичный ключ из таблицы <code>users</code> .                      |
| <code>\$table-&gt;dropUnique('users_email_unique');</code>              | Удалить уникальный индекс из таблицы <code>users</code> .                   |
| <code>\$table-&gt;dropIndex('geo_state_index');</code>                  | Удалить простой индекс из таблицы <code>geo</code> .                        |
| <code>\$table-&gt;dropFullText('posts_body_fulltext');</code>           | Удалить полнотекстовый индекс из таблицы <code>posts</code> .               |
| <code>\$table-&gt;dropSpatialIndex('geo_location_spatialindex');</code> | Удалить пространственный индекс из таблицы <code>geo</code> (кроме SQLite). |

Если вы передадите массив столбцов в метод, удаляющий индексы, то обычное имя индекса будет сгенерировано на основе имени таблицы, столбцов и типа индекса:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Удалить простой индекс `geo_state_index`.
});
```

## Ограничения внешнего ключа

Laravel также поддерживает создание ограничений внешнего ключа, которые используются для обеспечения ссылочной целостности на уровне базы данных. Например, давайте определим столбец `user_id` в таблице `posts`, который ссылается на столбец `id` в таблице `users`:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');
```

```
$table->foreign('user_id')->references('id')->on('users');  
});
```

Поскольку этот синтаксис довольно подробный, Laravel предлагает дополнительные, более сжатые методы, использующие соглашения, для повышения продуктивности разработки. При использовании метода `foreignId` для создания вашего столбца приведенный выше пример можно переписать так:

```
Schema::table('posts', function (Blueprint $table) {  
    $table->foreignId('user_id')->constrained();  
});
```

Метод `foreignId` создает столбец эквивалентный `UNSIGNED BIGINT`, в то время как метод `constrained` будет использовать соглашения для определения таблицы и столбца, на которые ссылаются. Если имя вашей таблицы не соответствует соглашениям Laravel, вы можете вручную указать его в методе `constrained`. Кроме того, можно также указать имя, которое должно быть присвоено созданному индексу:

```
Schema::table('posts', function (Blueprint $table) {  
    $table->foreignId('user_id')->constrained(  
        table: 'users', indexName: 'posts_user_id'  
    );  
});
```

Вы также можете указать желаемое действие для свойств ограничения «при удалении» и «при обновлении»:

```
$table->foreignId('user_id')  
    ->constrained()  
    ->onUpdate('cascade')  
    ->onDelete('cascade');
```

Для этих действий также предусмотрен альтернативный синтаксис выражений:

| <b>Метод</b>                                 | <b>Описание</b>  |
|--|--|
| <code>\$table-&gt;cascadeOnUpdate();</code>  | Обновления должны быть каскадными.                                   |
| <code>\$table-&gt;restrictOnUpdate();</code> | Обновления должны быть ограничены.                                   |
| <code>\$table-&gt;&gt;nullOnUpdate();</code> | Обновления должны устанавливать значение внешнего ключа null.        |
| <code>\$table-&gt;noActionOnUpdate();</code> | Ниаких действий по обновлениям.                                      |
| <code>\$table-&gt;cascadeonDelete();</code>  | Удаления должны быть каскадными.                                     |
| <code>\$table-&gt;restrictonDelete();</code> | Удаления должны быть ограничены.                                     |
| <code>\$table-&gt;&gt;nullonDelete();</code> | Для удаления следует установить значение внешнего ключа равным null. |
| <code>\$table-&gt;noActiononDelete();</code> | Предотвращает удаление, если существуют дочерние записи.             |

Любые дополнительные [модификаторы столбца](#) должны быть вызваны перед методом `constrained`:

```
$table->foreignId('user_id')
    ->nullable()
    ->constrained();
```

## Удаление внешних ключей

Чтобы удалить внешний ключ, вы можете использовать метод `dropForeign`, передав в качестве аргумента имя ограничения внешнего ключа, которое нужно удалить. Ограничения внешнего ключа используют то же соглашение об именах, что и

индексы. Другими словами, имя ограничения внешнего ключа основано на имени таблицы и столбцов в ограничении, за которым следует суффикс `_foreign`:

```
$table->dropForeign('posts_user_id_foreign');
```

В качестве альтернативы вы можете передать массив, содержащий имя столбца, который содержит внешний ключ, методу `dropForeign`. Массив будет преобразован в имя ограничения внешнего ключа с использованием соглашений об именах ограничений Laravel:

```
$table->dropForeign(['user_id']);
```

## Переключение ограничений внешнего ключа

Вы можете включить или отключить ограничения внешнего ключа в своих миграциях, используя следующие методы:

```
Schema::enableForeignKeyConstraints();  
  
Schema::disableForeignKeyConstraints();  
  
Schema::withoutForeignKeyConstraints(function () {  
    // Constraints disabled within this closure...  
});
```

SQLite по умолчанию отключает ограничения внешнего ключа. При использовании SQLite убедитесь, что [включили поддержку внешнего ключа](#) в вашей конфигурации базы данных, прежде чем пытаться создать их в ваших миграциях.

## # События

Для удобства каждая операция миграции отправляет [событие](#). Все следующие события расширяют базовый класс `Illuminate\Database\Events\MigrationEvent`:

| Класс   | Описание   |
|---|--|
| <code>Illuminate\Database\Events\MigrationsStarted</code>   | Вот-вот будет выполнен пакет миграций.             |
| <code>Illuminate\Database\Events\MigrationsEnded</code>     | Завершено выполнение пакета миграций.              |
| <code>Illuminate\Database\Events\MigrationStarted</code>    | Одна миграция вот-вот будет выполнена.             |
| <code>Illuminate\Database\Events\MigrationEnded</code>      | Выполнение одной миграции завершено.               |
| <code>Illuminate\Database\Events\NoPendingMigrations</code> | Команда миграции не обнаружила ожидающих миграций. |
| <code>Illuminate\Database\Events\SchemaDumped</code>        | Завершена выгрузка схемы базы данных.              |
| <code>Illuminate\Database\Events\SchemaLoaded</code>        | Загружена существующая выгрузка схемы базы данных. |

# База данных · Наполнение фиктивными данными

- # Введение
- # Написание наполнителей
  - # Использование фабрик моделей
  - # Вызов дополнительных наполнителей
  - # Отключение событий модели
- # Запуск наполнителей

## # Введение

Laravel предлагает возможность наполнения вашей базы тестовыми данными с использованием классов-наполнителей. Все классы наполнителей хранятся в каталоге `database/seeders`. Класс `DatabaseSeeder` уже определен по умолчанию. В этом классе вы можете использовать метод `call` для запуска других наполнителей, что позволит вам контролировать порядок наполнения БД.

При наполнении базы данных автоматически отключается защита [массового присвоения](#).

## # Написание наполнителей

Чтобы сгенерировать новый наполнитель, используйте команду `make:seeder Artisan`. Эта команда поместит новый класс наполнителя в каталог `database/seeders` вашего приложения:

```
php artisan make:seeder UserSeeder
```

Класс наполнителя (сидера) по умолчанию содержит только один метод: `run`. Этот метод вызывается при выполнении [команды Artisan db:seed](#). Внутри метода `run` вы можете вставлять данные в свою базу данных так, как вам удобно. Вы можете использовать [строитель запросов \(query builder\)](#) для ручной вставки данных или [фабрики моделей Eloquent](#).

В качестве примера давайте изменим класс `DatabaseSeeder`, созданный по умолчанию, и добавим выражение вставки фасада `DB` в методе `run`:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Запустить наполнение базы данных.
     */
    public function run(): void
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10). '@example.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

В методе `run` вы можете объявить любые необходимые типы зависимостей. Они будут автоматически извлечены и внедрены через [контейнер служб Laravel](#).

## Использование фабрик моделей

Конечно, ручное указание атрибутов для каждой модели наполнителя обременительно. Вместо этого вы можете использовать [фабрики моделей](#) для удобного создания большого количества записей в БД. Сначала просмотрите [документацию фабрики моделей](#), чтобы узнать, как определить свои фабрики.

Например, давайте создадим 50 пользователей, у каждого из которых будет по одному посту:

```
use App\Models\User;

/**
 * Запустить наполнение базы данных.
 */
public function run(): void
{
    User::factory()
        ->count(50)
        ->hasPosts(1)
        ->create();
}
```

## Вызов дополнительных наполнителей

Внутри класса `DatabaseSeeder` вы можете использовать метод `call` для запуска других наполнителей. Использование метода `call` позволяет вам разбить ваши наполнители БД на несколько файлов, так что ни один класс наполнителя не станет слишком большим. Метод `call` принимает массив классов, которые должны быть выполнены:

```
/**
 * Запустить наполнение базы данных.
 */
public function run(): void
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

## Отключение событий модели

При выполнении сидов (seeds) вы можете захотеть предотвратить моделям отправку событий. Для этого вы можете использовать трейт `WithoutModelEvents`. При его использовании, трейт `WithoutModelEvents` гарантирует, что события модели не будут отправлены, даже если дополнительные сид-классы выполняются с помощью метода `call`:

```
<?php

namespace Database\Seeders;

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

class DatabaseSeeder extends Seeder
{
    use WithoutModelEvents;

    /**
     * Запуск сидеров базы данных.
     */
    public function run(): void
    {
        $this->call([
            UserSeeder::class,
        ]);
    }
}
```

Этот трейт поможет вам отключить отправку событий модели во время выполнения сидов (seeds).

## # Запуск наполнителей

Вы можете выполнить команду `db:seed` Artisan для наполнения вашей базы данных. По умолчанию команда `db:seed` запускает класс `Database\Seeders\DatabaseSeeder`, который, в свою очередь, может вызывать другие классы. Однако вы можете использовать параметр `--class`, чтобы указать конкретный класс наполнителя для его индивидуального запуска:

```
php artisan db:seed
```

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

Вы также можете заполнить свою базу данных, используя команду `migrate:fresh` в сочетании с опцией `--seed`, которая удалит все таблицы и перезапустит все миграции. Эта команда полезна для полной перестройки вашей базы данных. Опцию `--seeder` можно использовать для указания конкретного сида (seeder) для выполнения:

```
php artisan migrate:fresh --seed --seeder=UserSeeder
```

## Принудительное наполнение при эксплуатации приложения

Некоторые операции наполнения могут привести к изменению или потере данных. В окружении `production`, чтобы защитить вас от запуска команд наполнения эксплуатируемой базы данных, вам будет предложено подтвердить их запуск. Чтобы заставить наполнители запускаться без подтверждений, используйте флаг `--force`:

```
php artisan db:seed --force
```

# База данных · Использование Redis

# Введение

# Конфигурирование

# Кластеры

# Predis

# PhpRedis

# Взаимодействие с Redis

# Транзакции

# Конвейерное выполнение команд

# Публикация / подписка

## # Введение

[Redis](#) – это расширенное хранилище ключ-значение с открытым исходным кодом. Его часто называют сервером структуры данных, поскольку ключи могут содержать [строки](#), [хеши](#), [списки](#), [наборы](#) и [отсортированные наборы](#).

Перед использованием Redis с Laravel мы рекомендуем вам установить и использовать расширение [PhpRedis](#) PHP через PECL. Расширение сложнее установить по сравнению с пакетами PHP пользовательского слоя, но оно может обеспечить лучшую производительность для приложений, интенсивно использующих Redis. Если вы используете [Laravel Sail](#), то это расширение уже установлено в контейнере Docker вашего приложения.

Если вы не можете установить расширение PhpRedis, то установите пакет [predis/predis](#) через Composer. Predis – это клиент Redis, полностью написанный на PHP и не требующий дополнительных расширений:

```
composer require predis/predis:^2.0
```

## # Конфигурирование

Вы можете настроить параметры Redis для своего приложения с помощью конфигурационного файла `config/database.php`. В этом файле вы увидите массив `redis`, содержащий серверы Redis, используемые вашим приложением:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'phpredis'),  
  
    'options' => [  
        'cluster' => env('REDIS_CLUSTER', 'redis'),  
        'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_'),  
    ],  
  
    'default' => [  
        'url' => env('REDIS_URL'),  
        'host' => env('REDIS_HOST', '127.0.0.1'),  
        'username' => env('REDIS_USERNAME'),  
        'password' => env('REDIS_PASSWORD'),  
        'port' => env('REDIS_PORT', '6379'),  
        'database' => env('REDIS_DB', '0'),  
    ],  
  
    'cache' => [  
        'url' => env('REDIS_URL'),  
        'host' => env('REDIS_HOST', '127.0.0.1'),  
        'username' => env('REDIS_USERNAME'),  
        'password' => env('REDIS_PASSWORD'),  
        'port' => env('REDIS_PORT', '6379'),  
        'database' => env('REDIS_CACHE_DB', '1'),  
    ],  
  
],  

```

Каждый сервер Redis, определенный в вашем конфигурационном файле, должен иметь имя, хост и порт, либо единый URL соединения Redis:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'phpredis'),  
  
    'options' => [  
        'cluster' => env('REDIS_CLUSTER', 'redis'),  
        'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_'),  
    ],  
  
],
```

```
'default' => [
    'url' => 'tcp://127.0.0.1:6379?database=0',
],
['cache' => [
    'url' => 'tls://user:password@127.0.0.1:6380?database=1',
],
],
],
```

## Настройка схемы подключения

По умолчанию клиенты Redis будут использовать схему `tcp` при подключении к вашим серверам Redis; однако вы можете использовать шифрование TLS / SSL, указав параметр `scheme` конфигурации в массиве конфигурации вашего сервера Redis:

```
< >
'default' => [
    'scheme' => 'tls',
    'url' => env('REDIS_URL'),
    'host' => env('REDIS_HOST', '127.0.0.1'),
    'username' => env('REDIS_USERNAME'),
    'password' => env('REDIS_PASSWORD'),
    'port' => env('REDIS_PORT', '6379'),
    'database' => env('REDIS_DB', '0'),
],
]
```

## Кластеры

Если ваше приложение использует кластер серверов Redis, то вы должны определить эти кластеры в ключе `clusters` вашей конфигурации Redis. Этот ключ конфигурации не существует по умолчанию, поэтому вам нужно будет создать его в конфигурационном файле `config/database.php` вашего приложения:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
        'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_'),
    ],
],
```

```
'clusters' => [
    'default' => [
        [
            'url' => env('REDIS_URL'),
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'username' => env('REDIS_USERNAME'),
            'password' => env('REDIS_PASSWORD'),
            'port' => env('REDIS_PORT', '6379'),
            'database' => env('REDIS_DB', '0'),
        ],
    ],
],
],
// ...
],
```

По умолчанию Laravel будет использовать встроенное кластерирование Redis, так как значение конфигурации `options.cluster` установлено на `redis`. Кластеризация Redis – отличный вариант по умолчанию, так как она гармонично обрабатывает аварийные ситуации.

Laravel также поддерживает клиентское разделение данных (sharding). Однако клиентское разделение данных не обрабатывает аварийные ситуации, поэтому оно в основном подходит для временных кешированных данных, доступных из другого основного хранилища данных.

Если вы хотите использовать клиентское разделение данных вместо встроенной кластеризации Redis, вы можете удалить значение конфигурации `options.cluster` в файле конфигурации вашего приложения `config/database.php`:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'clusters' => [
        // ...
    ],
],
// ...
],
```

## Predis

Если вы хотите, чтобы ваше приложение взаимодействовало с Redis через пакет Predis, то вы должны убедиться, что значение переменной окружения `REDIS_CLIENT` установлено как `predis`:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'predis'),  
  
    // ...  
,
```

Помимо параметров конфигурации по умолчанию, Predis поддерживает дополнительные [параметры подключения](#), которые могут быть определены для каждого из ваших серверов Redis. Чтобы использовать эти дополнительные параметры конфигурации, добавьте их в конфигурацию вашего сервера Redis в файле конфигурации вашего приложения `config/database.php`:

```
'default' => [  
    'url' => env('REDIS_URL'),  
    'host' => env('REDIS_HOST', '127.0.0.1'),  
    'username' => env('REDIS_USERNAME'),  
    'password' => env('REDIS_PASSWORD'),  
    'port' => env('REDIS_PORT', '6379'),  
    'database' => env('REDIS_DB', '0'),  
    'read_write_timeout' => 60,  
,
```

## PhpRedis

По умолчанию Laravel будет использовать расширение PhpRedis для соединения с Redis. Клиент, который Laravel будет использовать для соединения с Redis, определяется значением параметра `redis.client` конфигурации, который обычно проксирует значение переменной `REDIS_CLIENT` окружения:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'phpredis'),
```

```
// ...  
],
```

Помимо параметров конфигурации по умолчанию, PhpRedis поддерживает следующие дополнительные параметры подключения: `name`, `persistent`, `persistent_id`, `prefix`, `read_timeout`, `retry_interval`, `timeout` и `context`. Вы можете добавить любой из этих параметров в конфигурацию вашего сервера Redis в файле конфигурации вашего приложения `config/database.php`:

```
'default' => [  
    'url' => env('REDIS_URL'),  
    'host' => env('REDIS_HOST', '127.0.0.1'),  
    'username' => env('REDIS_USERNAME'),  
    'password' => env('REDIS_PASSWORD'),  
    'port' => env('REDIS_PORT', '6379'),  
    'database' => env('REDIS_DB', '0'),  
    'read_timeout' => 60,  
    'context' => [  
        // 'auth' => ['username', 'secret'],  
        // 'stream' => ['verify_peer' => false],  
    ],  
],
```

## PhpRedis Сериализация и сжатие

Расширение PhpRedis также можно настроить для использования различных алгоритмов сериализации и сжатия. Эти алгоритмы можно настроить с помощью массива `options` вашей конфигурации Redis:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'phpredis'),  
  
    'options' => [  
        'cluster' => env('REDIS_CLUSTER', 'redis'),  
        'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_'),  
        'serializer' => Redis::SERIALIZER_MSGPACK,  
        'compression' => Redis::COMPRESSION_LZ4,  
    ],  
  
    // ...  
],
```



В настоящее время поддерживаются следующие сериализаторы:

`Redis::SERIALIZER_NONE` (default), `Redis::SERIALIZER_PHP`, `Redis::SERIALIZER_JSON`,  
`Redis::SERIALIZER_IGBINARY`, и `Redis::SERIALIZER_MSGPACK`.

Поддерживаемые алгоритмы сжатия: `Redis::COMPRESSION_NONE` (default),  
`Redis::COMPRESSION_LZF`, `Redis::COMPRESSION_ZSTD`, и `Redis::COMPRESSION_LZ4`.

## # Взаимодействие с Redis

Вы можете взаимодействовать с Redis, вызывая различные методы [фасада Redis](#). Фасад `Redis` поддерживает динамические методы, то есть вы можете вызвать любую [команду Redis](#), используя фасад, и команда будет передана непосредственно в Redis. В этом примере мы вызовем команду Redis `GET`, вызвав метод `get` фасада `Redis`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Redis;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Показать профиль конкретного пользователя.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => Redis::get('user:profile:'.$id)
        ]);
    }
}
```

Как упоминалось выше, вы можете вызывать любую из команд Redis, используя фасад `Redis`. Laravel использует магические методы для передачи команд на сервер Redis. Если команда Redis ожидает аргументов, то вы должны передать их соответствующему методу фасада:

```
use Illuminate\Support\Facades\Redis;

Redis::set('name', 'Taylor');

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

В качестве альтернативы вы можете передавать команды серверу, используя метод `command` фасада `Redis`, который принимает имя команды в качестве первого аргумента и массив значений в качестве второго аргумента:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

## Использование нескольких подключений Redis

Конфигурационный файл `config/database.php` вашего приложения позволяет вам определять несколько соединений / серверов Redis. Вы можете получить соединение с конкретным соединением Redis, используя метод `connection` фасада `Redis`:

```
$redis = Redis::connection('connection-name');
```

Чтобы получить экземпляр соединения Redis по умолчанию, вы можете вызвать метод `connection` без каких-либо дополнительных аргументов:

```
$redis = Redis::connection();
```

## Транзакции

Метод `transaction` фасада `Redis` обеспечивает удобную обертку для собственных команд `MULTI` и `EXEC` Redis. Метод `transaction` принимает замыкание как единственный аргумент. Это замыкание получит экземпляр подключения Redis и может использовать любые необходимые вам команды, отправляемые на сервер Redis. Все команды Redis в рамках замыкания будут выполняться в одной атомарной транзакции:

```
use Redis;
use Illuminate\Support\Facades;
```

```
Facades\\Redis::transaction(function (Redis $redis) {  
    $redis->incr('user_visits', 1);  
    $redis->incr('total_visits', 1);  
});
```

При определении транзакции Redis вы не можете получать какие-либо значения из соединения Redis. Помните, ваша транзакция выполняется как одна атомарная операция, и эта операция не выполнится, пока не завершится выполнение всех команд замыкания.

## Скрипты Lua

Метод `eval` обеспечивает другой метод выполнения нескольких команд Redis за одну атомарную операцию. Однако преимущество метода `eval` состоит в том, что он может взаимодействовать со значениями ключей Redis и использовать их во время этой операции. Скрипты Redis написаны на [языке программирования Lua](#).

Поначалу метод `eval` может показаться немного пугающим, но мы рассмотрим пример. Метод `eval` ожидает несколько аргументов. Во-первых, вы должны передать сценарий Lua (в виде строки) в метод. Во-вторых, вы должны передать количество ключей (в виде целого числа), с которыми скрипт взаимодействует. В-третьих, вы должны передать имена этих ключей. Наконец, вы можете передать любые другие дополнительные аргументы, к которым вам нужно получить доступ в вашем скрипте.

В этом примере мы увеличим счетчик, проверим его новое значение и увеличим второй счетчик, если значение первого счетчика больше пяти. Наконец, мы вернем значение первого счетчика:

```
$value = Redis::eval(<<<'LUA'  
local counter = redis.call("incr", KEYS[1])  
  
if counter > 5 then  
    redis.call("incr", KEYS[2])  
end
```

```
    return counter
LUA, 2, 'first-counter', 'second-counter');
```

Пожалуйста, обратитесь к [документации Redis](#) для получения дополнительных сведений о сценариях Redis.

## Конвейерное выполнение команд

По желанию можно выполнить десятки команд Redis. Вместо того чтобы совершать сетевое обращение к вашему серверу Redis для каждой команды, вы можете использовать метод `pipeline`. Метод `pipeline` принимает один аргумент: замыкание, которое получает экземпляр Redis. Вы можете передать все свои команды этому экземпляру Redis, и все они будут отправлены на сервер Redis одновременно, чтобы уменьшить количество сетевых обращений к серверу. Команды по-прежнему будут выполняться в том порядке, в котором они были отправлены:

```
use Redis;
use Illuminate\Support\Facades;

Facades\Redis::pipeline(function (Redis $pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

## # Публикация / подписка

Laravel предлагает удобный интерфейс для команд `publish` и `subscribe` Redis. Эти команды Redis позволяют вам прослушивать сообщения на указанном «канале». Вы можете публиковать сообщения в канал из другого приложения или даже с использованием другого языка программирования, что позволяет легко взаимодействовать между приложениями и процессами.

Во-первых, давайте настроим слушатель каналов с помощью метода `subscribe`. Мы поместим вызов этого метода в [команду Artisan](#), поскольку вызов метода `subscribe` запускает длительный процесс:

```
<?php
```

```
namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * Имя и сигнатура консольной команды.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * Описание консольной команды.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Выполнить консольную команду.
     */
    public function handle(): void
    {
        Redis::subscribe(['test-channel'], function (string $message) {
            echo $message;
        });
    }
}
```

Теперь мы можем публиковать сообщения в канале с помощью метода `publish`:

```
use Illuminate\Support\Facades\Redis;

Route::get('/publish', function () {
    // ...

    Redis::publish('test-channel', json_encode([
        'name' => 'Adam Wathan'
    ]));
});
```

## Групповые подписки

Допускается использование метасимвола подстановки `*` при использовании метода `psubscribe`, что позволит вам перехватывать все сообщения на нескольких каналах. Имя канала будет передано вторым аргументом в указанное замыкание:

```
Redis::psubscribe(['*'], function (string $message, string $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function (string $message, string $channel) {
    echo $message;
});
```

# Eloquent · Начало работы

## # Введение

### # Генерация классов модели

### # Соглашения по именованию моделей Eloquent

# Именование таблиц

# Первичные ключи

# UUID и ULID ключи

# Временные метки

# Соединения с БД

# Значения атрибутов по умолчанию

# Настройка строгости Eloquent

## # Получение моделей

# Коллекции

# Разбиение результатов

# Разбиение на части с использованием ленивых коллекций

# Курсоры

# Расширенные подзапросы

## # Извлечение отдельных моделей

# Получение или создание моделей

# Извлечение Агрегатов

## # Вставка и обновление моделей

# Вставка

# Обновление

# Массовое присвоение

# Исключения при массовом присвоении

# Обновления-вставки

## # Удаление моделей

# Программное удаление

# Запросы для моделей, использующих программное удаление

## # Периодическое удаление (pruning) старых записей

# Репликация (тиражирование) моделей

# Диапазоны запроса (scopes)

# Глобальные диапазоны

# Локальные диапазоны

# Сравнение моделей

# События

# Использование замыканий

# Наблюдатели

# Подавление событий

## # Введение

Laravel содержит [ORM-библиотеку](#) Eloquent, предоставляющую способ работы с базой данных, который часто удобнее обычного построителя запросов. При использовании Eloquent каждая таблица БД имеет соответствующую «Модель», которая используется для взаимодействия с этой таблицей. Помимо получения записей из таблицы БД, модели Eloquent также позволяют вставлять, обновлять и удалять записи из таблицы.

Перед началом работы настройте соединение с БД в конфигурационном файле [config/database.php](#). Для получения дополнительной информации о настройке БД ознакомьтесь с [документацией по конфигурированию БД](#).

## Курс по Laravel (Laravel Bootcamp)

Если вы новичок в Laravel, не стесняйтесь присоединиться к [Laravel Bootcamp](#). Laravel Bootcamp поможет вам создать свое первое приложение Laravel с использованием Eloquent. Это отличный способ получить обзор всего, что предлагают Laravel и Eloquent.

## # Генерация классов модели

Модели расширяют класс `Illuminate\Database\Eloquent\Model`. Чтобы сгенерировать новую модель Eloquent, используйте [Artisan](#)-команду `make:model`. Эта команда поместит новый класс модели в каталог `app/Models` вашего приложения:

```
php artisan make:model Flight
```

При создании модели вы можете сгенерировать [миграцию БД](#), используя параметр `--migration` или `-m`:

```
php artisan make:model Flight --migration
```

При создании модели вы можете также создавать другие различные типы классов, например фабрики, наполнители (seeders), политики, контроллеры и запросы форм. Кроме того, эти параметры можно комбинировать для создания сразу нескольких классов:

```
# Создать модель и класс FlightFactory ...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Создать модель и класс FlightSeeder...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Создать модель и класс FlightController...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Создать модель, класс ресурса FlightController и класс запроса формы...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Создать модель и класс FlightPolicy class...
php artisan make:model Flight --policy

# Создать модель, миграцию, фабрику, наполнитель и контроллер ...
php artisan make:model Flight -mfsc

# Создать модель, миграцию, фабрику, наполнитель, политику, контроллер и запрос формы
php artisan make:model Flight --all
php artisan make:model Flight -a
```

```
# Создать сводную модель...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

## Обзор моделей

Иногда бывает сложно определить все доступные атрибуты и отношения модели, просто просматривая ее код. Вместо этого попробуйте использовать команду Artisan `model:show`, которая предоставляет удобный обзор всех атрибутов и отношений модели:

```
php artisan model:show Flight
```

## # Соглашения по именованию моделей Eloquent

Модели, созданные командой `make:model`, будут помещены в каталог `app/Models`.

Давайте рассмотрим базовый класс модели и обсудим некоторые ключевые соглашения Eloquent:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    // ...
}
```

## Именование таблиц

Взглянув на приведенный выше пример, вы могли заметить, что мы не сообщили Eloquent, какая таблица БД соответствует нашей модели `Flight`. По соглашению, в качестве имени таблицы будет использоваться имя класса в формате `snake_case` (все), во множественном числе, если явно не указано другое. В нашем случае, Eloquent будет предполагать, что модель `Flight` хранит записи в таблице `flights`, а модель `AirTrafficController` – в таблице `air_traffic_controllers`.

Если таблица БД вашей модели не соответствует этому соглашению, вы можете вручную указать имя таблицы модели, определив свойство `table` в модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Таблица БД, ассоциированная с моделью.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

## Первичные ключи

Eloquent также предполагает, что в соответствующей таблице БД каждой модели есть столбец первичного ключа с именем `id`. При необходимости вы можете определить защищенное свойство `$primaryKey` в модели, чтобы указать другой столбец, который служит первичным ключом:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Первичный ключ таблицы БД.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

Кроме того, Eloquent предполагает, что первичный ключ является автоинкрементным целочисленным значением, что означает, что Eloquent автоматически преобразует первичный ключ в целое число. Если вы хотите использовать неинкрементный или нечисловой первичный ключ, вы должны определить общедоступное свойство `$incrementing` в модели, для которого установлено значение `false`:

```
<?php

class Flight extends Model
{
    /**
     * Указывает, что идентификаторы модели являются автоинкрементными.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

Если первичный ключ модели не является целочисленным, то определите защищенное свойство `$keyType` в модели. Это свойство имеет значение типа `string`:

```
<?php

class Flight extends Model
{
    /**
     * Тип данных автоинкрементного идентификатора.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

## «Составные» первичные ключи

Eloquent требует, чтобы каждая модель имела по крайней мере один однозначно идентифицирующий «ID», который может служить ее первичным ключом. «Составные» первичные ключи не поддерживаются моделями Eloquent. Однако вы можете добавить дополнительные многоколоночные уникальные индексы к таблицам базы данных в дополнение к однозначно определяющему (уникальному) первичному ключу таблицы.

## UUID и ULID ключи

Вместо использования автоинкрементных целых чисел в качестве первичных ключей вашей модели Eloquent вы можете выбрать использование UUID. UUID – это уникальные буквенно-цифровые идентификаторы длиной 36 символов.

Если вы хотите, чтобы модель использовала ключ UUID вместо автоинкрементного целочисленного ключа, вы можете использовать трейт [Illuminate\Database\Eloquent\Concerns\HasUuids](#) в модели. Конечно же, убедитесь, что у модели есть [столбец первичного ключа, эквивалентный UUID](#):

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUuids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Europe']);

$article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

По умолчанию трейт [HasUuids](#) генерирует [“упорядоченные” UUID](#) для ваших моделей. Эти UUID более эффективны для индексированного хранения в базе данных, поскольку их можно лексикографически сортировать.

Вы можете переопределить процесс генерации UUID для определенной модели, определив метод [newUniqueId](#) в модели. Кроме того, вы можете указать, какие столбцы должны получать UUID, определив метод [uniqueIds](#) в модели:

```
use Ramsey\Uuid\Uuid;

/**
 * Generate a new UUID for the model.
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}
```

```

/**
 * Get the columns that should receive a unique identifier.
 *
 * @return array<int, string>
 */
public function uniqueIds(): array
{
    return ['id', 'discount_code'];
}

```

Если вы хотите, вы можете вместо UUID использовать "ULID". ULID аналогичны UUID, однако они имеют длину всего 26 символов. Как и у упорядоченных UUID, ULID лексикографически сортируются для эффективного индексирования в базе данных. Для использования ULID вы должны использовать трейт [Illuminate\Database\Eloquent\Concerns\HasUlids](#) в вашей модели. Также убедитесь, что у модели есть столбец первичного ключа, эквивалентный ULID:

```

use Illuminate\Database\Eloquent\Concerns\HasUlids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUlids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Asia']);

$article->id; // "01gd4d3tgrrfqeda94gdbtdk5c"

```

## Временные метки

По умолчанию Eloquent ожидает, что столбцы `created_at` и `updated_at` будут существовать в соответствующей таблице БД модели. Eloquent автоматически устанавливает значения этих столбцов при создании или обновлении моделей. Если вы не хотите, чтобы эти столбцы автоматически управлялись Eloquent, вы должны определить свойство `$timestamps` модели со значением `false`:

```

<?php

namespace App\Models;

```

```
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Следует ли обрабатывать временные метки модели.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

Если вам нужно настроить формат временных меток модели, то укажите необходимый формат для свойства `$dateFormat`. Это свойство определяет, как атрибуты даты хранятся в БД, а также их формат при сериализации модели в массив или JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Формат хранения столбцов даты модели.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Если вам нужно настроить имена столбцов, используемых для хранения временных меток, то укажите значения для констант `CREATED_AT` и `UPDATED_AT` в модели:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
```

```
    const UPDATED_AT = 'updated_date';
}
```

Если вы хотите выполнять операции с моделью, не изменяя ее метку времени `updated_at`, вы можете выполнять операции с моделью внутри замыкания, переданного методу `withoutTimestamps`:

```
Model::withoutTimestamps(fn () => $post->increment('reads'));
```

## Соединения с БД

По умолчанию все модели Eloquent будут использовать соединение с БД, настроенное для вашего приложения. Если вы хотите указать другое соединение, которое должно использоваться при взаимодействии с определенной моделью, вы должны определить свойство `$connection` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Соединение с БД, которое должна использовать модель.
     *
     * @var string
     */
    protected $connection = 'mysql';
}
```

## Значения атрибутов по умолчанию

По умолчанию вновь созданный экземпляр модели не будет содержать никаких значений атрибутов. Если вы хотите определить значения по умолчанию для некоторых атрибутов модели, то укажите необходимые значения в свойстве `$attributes` модели. Значения атрибутов, помещенные в массив `$attributes`, должны быть в их исходном, "хранящемся" формате, как если бы они только что были считаны из базы данных::

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Значения по умолчанию для атрибутов модели.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

## Настройка строгости Eloquent

Laravel предоставляет несколько методов, которые позволяют настраивать поведение Eloquent и “строгость” в различных ситуациях.

Во-первых, метод `preventLazyLoading` принимает необязательный булевый аргумент, указывающий, следует ли запретить отложенную загрузку. Например, вы можете решить отключить отложенную загрузку только в не-продакшн средах, чтобы ваша продакшн среда продолжала функционировать normally, даже если отложенная загрузка отношения случайно присутствует в рабочем коде. Обычно этот метод следует вызывать в методе `boot` сервис-провайдера (`AppServiceProvider`) вашего приложения:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Запустите любые службы приложения.
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

Также вы можете указать Laravel генерировать исключение при попытке назначения неподлежащего назначению атрибута, вызвав метод `preventSilentlyDiscardingAttributes`. Это может помочь предотвратить неожиданные ошибки во время локальной разработки, когда попытаетесь присвоить значение, который не был добавлен в массив `fillable` модели:

```
Model::preventSilentlyDiscardingAttributes(! $this->app->isProduction());
```

## # Получение моделей

Создав модель и [связанную с ней таблицу БД](#), попробуем получить данные из БД при помощи модели. Вы должны думать о каждой модели Eloquent как о мощном [построителе запросов](#), позволяющем свободно выполнять запросы к таблице БД, связанной с моделью. Метод модели `all` получит все записи из связанной с моделью таблицы БД:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

## Создание запросов

Метод Eloquent `all` вернет все результаты из таблицы модели. Однако, поскольку каждая модель Eloquent служит [построителем запросов](#), вы можете добавить дополнительные условия к запросам, а затем вызвать метод `get` для получения результатов:

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();
```

Поскольку модель Eloquent является построителем запросов, вам следует просмотреть все методы,

предлагаемые [построителем запросов](#). Вы можете использовать любой из этих методов при написании запросов Eloquent.

## Обновление моделей

Если у вас уже есть экземпляр модели Eloquent, полученный из БД, вы можете «обновить» модель, используя методы `fresh` и `refresh`. Метод `fresh` повторно извлечет модель из БД. Существующий экземпляр модели не будет затронут:

```
$flight = Flight::where('number', 'FR 900')->first();  
  
$freshFlight = $flight->fresh();
```

Метод `refresh` повторно обновит существующую модель, используя свежие данные из БД. Кроме того, будут обновлены все загруженные отношения:

```
$flight = Flight::where('number', 'FR 900')->first();  
  
$flight->number = 'FR 456';  
  
$flight->refresh();  
  
$flight->number; // "FR 900"
```

## Коллекции

Как мы видели, методы Eloquent, такие как `all` и `get`, получают несколько записей из БД. Однако эти методы не возвращают простой массив PHP. Вместо этого возвращается экземпляр [Illuminate\Database\Eloquent\Collection](#).

Класс Eloquent `Collection` расширяет базовый класс Laravel [Illuminate\Support\Collection](#), который содержит [множество полезных методов](#) для взаимодействия с коллекциями данных. Например, метод `reject` используется для удаления моделей из коллекции на основе результатов вызванного замыкания:

```
$flights = Flight::where('destination', 'Paris')->get();
```

```
$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

Помимо методов, предоставляемых базовым классом коллекции Laravel, класс коллекции Eloquent содержит [несколько дополнительных методов](#), которые специально предназначены для взаимодействия с коллекциями моделей Eloquent.

Поскольку все коллекции Laravel реализуют итерируемые интерфейсы PHP, вы можете перебирать коллекции, как если бы они были массивом:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

## Разбиение результатов

Вашему приложению может не хватить памяти, если вы попытаетесь загрузить десятки тысяч записей Eloquent с помощью методов `all` или `get`. Вместо использования этих методов можно использовать метод `chunk` для более эффективной обработки большого количества моделей.

Метод `chunk` будет извлекать подмножество моделей Eloquent, передавая их в замыкание для обработки. Поскольку за один раз извлекается только текущая коллекция моделей Eloquent, метод `chunk` обеспечивает значительно меньшее потребление памяти при работе с большим количеством моделей:

```
use App\Models\Flight;
use Illuminate\Database\Eloquent\Collection;

Flight::chunk(200, function (Collection $flights) {
    foreach ($flights as $flight) {
        // ...
    }
});
```

Первым аргументом, передаваемым методу `chunk`, является количество записей, которые вы хотите получить за «порцию». Замыкание, переданное в качестве второго аргумента, будет вызвано для каждой части записей, полученной

из БД. Будет выполнен запрос к БД для получения каждой части записей, переданных замыканию.

Если вы фильтруете результаты метода `chunk` на основе столбца, который вы также будете обновлять при итерации результатов, вам следует использовать метод `chunkById`. Использование метода `chunk` в этих сценариях может привести к неожиданным и противоречивым результатам. Внутренне метод `chunkById` всегда будет извлекать модели со столбцом `id`, большим, чем у последней модели в предыдущей «порции»:

```
Flight::where('departed', true)
->chunkById(200, function (Collection $flights) {
    $flights->each->update(['departed' => false]);
}, column: 'id');
```

Поскольку методы `chunkById` и `lazyById` добавляют свои собственные условия “`where`” к выполняемому запросу, вам обычно следует логически группировать свои собственные условия внутри закрытия:

```
Flight::where(function ($query) {
    $query->where('delayed', true)->orWhere('cancelled', true);
})->chunkById(200, function (Collection $flights) {
    $flights->each->update([
        'departed' => false,
        'cancelled' => true
    ]);
}, column: 'id');
```

## Разбиение на части с использованием ленивых коллекций

Метод `lazy` работает аналогично [методу `chunk`](#) в том смысле, что он выполняет запрос по частям. Однако вместо передачи каждого фрагмента непосредственно в замыкание, метод `lazy()` возвращает экземпляр [LazyCollection](#) одноуровневых моделей Eloquent, что позволяет вам взаимодействовать с результатами как с единым потоком:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
```

```
// ...
}
```

Если вы фильтруете результаты метода `lazy` по столбцу, который впоследствии будет обновлен при итерации результатов, то вам следует использовать метод `lazyById`. Внутренне метод `lazyById` всегда будет извлекать модели со столбцом `id`, большим, чем у последней модели в предыдущей «порции»:

```
Flight::where('departed', true)
->lazyById(200, column: 'id')
->each->update(['departed' => false]);
```

Вы можете отфильтровать результаты по убыванию `id`, используя метод `lazyByIdDesc`.

## Курсыры

Подобно методу `lazy`, метод `cursor` используется для значительного уменьшения потребления памяти вашим приложением при итерации десятков тысяч записей модели Eloquent.

Метод `cursor` выполнит только один запрос к БД; однако отдельные модели Eloquent не будут включены в результирующий набор, пока они не будут фактически итерированы. Следовательно, только одна модель Eloquent хранится в памяти в любой момент времени при итерации с использованием курсора.

Поскольку метод `cursor` всегда хранит в памяти только одну модель Eloquent, то “жадная” (`eager`) загрузка отношений недопустима. Если вам нужно “жадно” загрузить отношения, то рассмотрите возможность использования метода `lazy`.

Внутри метод `cursor` использует [генераторы PHP](#) для реализации этого функционала:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    // ...
}
```

Курсор возвращает экземпляр [Illuminate\Support\LazyCollection](#). [Отложенные коллекции](#) позволяют использовать многие методы коллекций, доступные в типичных коллекциях Laravel, при одновременной загрузке в память только одной модели:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Хотя метод `cursor` использует гораздо меньше памяти, чем обычный запрос (удерживая в памяти только одну модель Eloquent), он все равно в конечном итоге может исчерпать память. Это связано с тем, что [драйвер PDO PHP внутренне кэширует все необработанные результаты запросов в своем буфере](#). Если вы имеете дело с очень большим количеством записей Eloquent, то рассмотрите возможность использования метода `lazy`.

## Расширенные подзапросы

### Выборка

Eloquent также предлагает поддержку расширенных подзапросов, которая позволяет извлекать информацию из связанных таблиц в одном запросе. Например, давайте представим, что у нас есть таблица `destinations` (пункты назначения) и `flights` (рейсы). В таблице `flights` содержится столбец `arrived_at`, который указывает, когда рейс прибыл в пункт назначения.

Используя функциональность подзапроса, доступную для методов `select` и `addSelect` построителя запросов, мы можем выбрать все `destinations` и название рейса, который последним прибыл в этот пункт назначения, используя один запрос:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

## Сортировка

Кроме того, метод `orderBy` построителя запросов поддерживает подзапросы. Продолжая использовать наш пример полетов, мы можем использовать этот метод для сортировки всех пунктов назначения в зависимости от того, когда последний рейс прибыл в этот пункт назначения. Опять же, это можно сделать при выполнении одного запроса к БД:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

## # Извлечение отдельных моделей

В дополнение к получению всех записей, соответствующих указанному запросу, вы также можете получить отдельные записи, используя методы `find`, `first` или `firstWhere`. Вместо того чтобы возвращать коллекцию моделей, эти методы возвращают единственный экземпляр модели:

```
use App\Models\Flight;

// Получить модель по ее первичному ключу ...
$flight = Flight::find(1);
```

```
// Получить первую модель, соответствующую условиям запроса ...
$flight = Flight::where('active', 1)->first();

// Альтернатива извлечению первой модели, соответствующей условиям запроса ...
$flight = App\Models\Flight::firstWhere('active', 1);
```

По желанию можно выполнить какое-либо другое действие, если результаты не найдены. Методы `findOr` и `firstOr` вернут один экземпляр модели или, если результаты не найдены, выполнят переданное замыкание. Значение, возвращенное замыканием, будет считаться результатом метода:

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

## Исключения при отсутствии результатов запроса

По желанию можно выбросить исключение, если модель не найдена. Это особенно полезно в маршрутах или контроллерах. Методы `findOrFail` и `firstOrFail` будут получать первый результат запроса; однако, если результат не найден, будет выброшено исключение `Illuminate\Database\Eloquent\ModelNotFoundException`:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

Если исключение не перехвачено, то клиенту автоматически отправляется HTTP-ответ `404`:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function (string $id) {
    return Flight::findOrFail($id);
});
```

# Получение или создание моделей

Метод `firstOrCreate` попытается найти запись в БД, используя указанные пары столбец / значение. Если модель не может быть найдена в БД, будет вставлена запись с атрибутами, полученными в результате объединения первого аргумента массива с необязательным вторым аргументом массива:

Метод `firstOrNew`, как и `firstOrCreate`, попытается найти в БД запись, соответствующую указанным атрибутам. Однако, если модель не найдена, будет возвращен новый экземпляр модели. Обратите внимание, что модель, возвращенная `firstOrNew`, еще не сохранена в БД. Вам нужно будет вручную вызвать метод `save`, чтобы сохранить его:

```
use App\Models\Flight;

// Получить рейс по `name` или создать его, если его не существует ...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Получить рейс по `name` или создать его с атрибутами `name`, `delayed` и `arrival_
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Получить рейс по `name` или создать новый экземпляр Flight ...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Получить рейс по `name` или создать экземпляр с атрибутами `name`, `delayed` и `ar
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

## Извлечение Агрегатов

При взаимодействии с моделями Eloquent вы также можете использовать `count`, `sum`, `max` и другие [агрегатные методы](#), предоставляемые [построителем запросов](#) Laravel. Как и следовало ожидать, эти методы возвращают соответствующее скалярное значение вместо экземпляра модели Eloquent:

```
$count = Flight::where('active', 1)->count();  
  
$max = Flight::where('active', 1)->max('price');
```

## # Вставка и обновление моделей

### Вставка

Конечно, при использовании Eloquent нам нужно не только извлекать модели из БД. Также нам нужно вставлять новые записи. К счастью, Eloquent делает это просто. Чтобы вставить новую запись в БД, вы должны создать экземпляр новой модели и установить атрибуты модели. Затем вызовите метод `save` экземпляра модели:

```
<?php
```

```
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\Models\Flight;  
use Illuminate\Http\RedirectResponse;  
use Illuminate\Http\Request;  
  
class FlightController extends Controller  
{  
    /**  
     * Сохранить новый рейс в базе данных.  
     */  
    public function store(Request $request): RedirectResponse  
    {  
        // Валидация запроса ...  
  
        $flight = new Flight;  
  
        $flight->name = $request->name;  
  
        $flight->save();  
  
        return redirect('/flights');  
    }  
}
```

В этом примере мы присваиваем параметр `name` из входящего HTTP-запроса атрибуту `name` экземпляра модели `App\Models\Flight`. Когда мы вызываем метод `save`, запись будет вставлена в БД. Временные метки `created_at` и `updated_at` будут автоматически установлены при вызове метода `save`, поэтому нет необходимости устанавливать их вручную.

В качестве альтернативы вы можете использовать метод `create`, чтобы «сохранить» новую модель с помощью одного оператора PHP. Вставленный экземпляр модели будет возвращен вам методом `create`:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Однако, перед использованием метода `create` вам нужно будет указать свойство `fillable` или `guarded` в классе модели. Эти свойства необходимы, потому что все модели Eloquent по умолчанию защищены от уязвимостей массового присвоения. Чтобы узнать больше о массовом присвоении, обратитесь к [документации](#).

## Обновление

Метод `save` также используется для обновления моделей, которые уже существуют в БД. Чтобы обновить модель, вы должны извлечь ее и установить любые атрибуты, которые вы хотите обновить. Затем вы должны вызвать метод `save`. Опять же, временная метка `updated_at` будет автоматически обновлена, поэтому нет необходимости вручную устанавливать ее значение:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Иногда вам может потребоваться обновить существующую модель или создать новую, если подходящей модели не существует. Как и метод `firstOrCreate`, метод

`updateOrCreate` сохраняет модель, поэтому нет необходимости вручную вызывать метод `save`.

В приведенном ниже примере, если существует `departure` (рейс) с местом отправления `Oakland` и `destination` (местом назначения) `San Diego`, его столбцы `price` (цена) и `discounted` (скидка) будут обновлены. Если такого полета не существует, будет создан новый полет с атрибутами, полученными в результате слияния первого массива аргументов со вторым массивом аргументов:

```
$flight = Flight::updateOrCreate(  
    ['departure' => 'Oakland', 'destination' => 'San Diego'],  
    ['price' => 99, 'discounted' => 1]  
)
```

## Массовые обновления

Обновления также могут выполняться для моделей, соответствующих указанному запросу. В этом примере все рейсы, которые активны и имеют пункт назначения в Сан-Диего, будут помечены как задержанные:

```
Flight::where('active', 1)  
    ->where('destination', 'San Diego')  
    ->update(['delayed' => 1]);
```

Метод `update` ожидает массив пар ключей и значений, представляющих столбцы, которые должны быть обновлены. Метод `update` возвращает количество затронутых строк.

События модели Eloquent `saving`, `saved`, `updating`, и `updated` при массовом обновлении **не будут инициированы** для затронутых моделей. Это связано с тем, что модели фактически никогда не извлекаются при массовом обновлении.

## Изучение изменений атрибутов

Eloquent содержит методы `isDirty`, `isClean` и `wasChanged` для проверки внутреннего состояния модели и определения того, как изменились ее атрибуты с момента первоначального извлечения модели.

Метод `isDirty` определяет, были ли изменены какие-либо атрибуты модели с момента получения модели. Вы можете передать конкретное имя атрибута или массив имён методу `isDirty`, чтобы определить, был ли изменен хотя бы один из этих атрибутов. Метод `isClean` определяет, остался ли атрибут неизменным с момента получения модели. Этот метод также принимает необязательный аргумент атрибута:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

Метод `wasChanged` определяет, были ли изменены какие-либо атрибуты при последнем сохранении модели в текущем цикле запроса. При необходимости вы можете передать имя атрибута, чтобы увидеть, был ли изменен конкретный атрибут:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
```

```
'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

Метод `getOriginal` возвращает массив, содержащий исходные атрибуты модели, независимо от каких-либо изменений в модели с момента ее получения. При необходимости вы можете передать конкретное имя атрибута, чтобы получить исходное значение определенного атрибута:

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

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Массив исходных атрибутов ...
```

## Массовое присвоение

Вы можете использовать метод `create`, чтобы «сохранить» новую модель с помощью одного оператора PHP. Вставленный экземпляр модели будет возвращен из метода:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Однако перед использованием метода `create` вам нужно будет указать свойство `fillable` или `guarded` в классе модели. Эти свойства необходимы, потому что все модели Eloquent по умолчанию защищены от уязвимостей массового присвоения.

Уязвимость массового присвоения возникает, когда пользователь передает неожиданное поле HTTP-запроса, и это поле изменяет столбец в вашей базе данных, чего вы никак не ожидали. Например, злоумышленник может отправить параметр `is_admin` через HTTP-запрос, который затем передается методу `create` модели, позволяя пользователю перейти на уровень администратора.

Итак, для начала вы должны определить, какие атрибуты модели вы хотите сделать массово-назначаемыми. Вы можете сделать это используя свойство `$fillable` модели. Например, давайте сделаем атрибут `name` нашей модели `Flight` массово-назначаемым:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Атрибуты, для которых разрешено массовое присвоение значений.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

После того как вы указали, какие атрибуты массово-назначаемые, вы можете использовать метод `create` для вставки новой записи в базу данных. Метод `create` возвращает вновь созданный экземпляр модели:

```
$flight = Flight::create(['name' => 'London to Paris']);
```

Если у вас уже есть экземпляр модели, вы можете использовать метод `fill` для заполнения его массивом атрибутов:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

## Массовое присвоение и JSON-столбцы

При назначении JSON-столбцов необходимо указать массово назначаемый ключ для каждого столбца в массиве `$fillable` модели. В целях безопасности Laravel не поддерживает обновление вложенных атрибутов JSON при использовании свойства `guarded`:

```
/**  
 * Атрибуты, для которых разрешено массовое присвоение значений.  
 *  
 * @var array  
 */  
protected $fillable = [  
    'options->enabled',  
];
```

## Защита массового присвоения

Если вы хотите, чтобы все ваши атрибуты были массово-назначаемыми, вы можете определить свойство модели `$guarded` как пустой массив. Если вы решите не защищать свою модель, вам следует позаботиться о том, чтобы всегда вручную обрабатывать массивы, переданные в методы Eloquent `fill`, `create` и `update`:

```
/**  
 * Атрибуты, для которых НЕ разрешено массовое присвоение значений.  
 *  
 * @var array  
 */  
protected $guarded = [];
```

## Исключения при массовом присвоении

По умолчанию атрибуты, которые не включены в массив `$fillable`, автоматически отбрасываются при выполнении операций массового присвоения. В продакшн-среде это ожидаемое поведение, однако во время локальной разработки это может вызвать путаницу в том, почему изменения модели не вступают в силу.

Вы можете указать Laravel генерировать исключение при попытке назначения неподлежащего назначения атрибута, вызвав метод `preventSilentlyDiscardingAttributes`. Обычно этот метод следует вызывать в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Запустите любые службы приложения.
 */
public function boot(): void
{
    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());
}
```

## Обновления-вставки

Метод `upsert` в Eloquent можно использовать для обновления или создания записей за одну атомарную операцию. Первый аргумент метода состоит из значений для вставки или обновления, а второй аргумент перечисляет столбцы, которые однозначно идентифицируют записи в связанной таблице. Третий и последний аргумент метода — это массив столбцов, который следует обновить, если соответствующая запись уже существует в базе данных. Метод `upsert` автоматически установит временные метки `created_at` и `updated_at`, если временные метки включены в модели:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], uniqueBy: ['departure', 'destination'], update: ['price']);
```

Все базы данных, кроме SQL Server, требуют, чтобы столбцы второго аргумента метода `upsert` имели “primary” или “unique” индекс. Кроме того, драйверы базы данных MariaDB и MySQL игнорируют второй аргумент метода `upsert` и всегда используют

“primary” и “unique” индексы таблицы для обнаружения существующих записей.

## # Удаление моделей

Чтобы удалить модель, вызовите метод `delete` экземпляра модели:

```
use App\Models\Flight;  
  
$flight = Flight::find(1);  
  
$flight->delete();
```

Вы можете вызвать метод `truncate`, чтобы удалить все записи базы данных, связанные с моделью. Операция `truncate` также сбрасывает все автоинкрементные идентификаторы в связанной с моделью таблице:

```
Flight::truncate();
```

### Удаление существующей модели по ее первичному ключу

В приведенном выше примере мы извлекаем модель из БД перед вызовом метода `delete`. Однако если вы знаете первичный ключ модели, вы можете удалить модель, не извлекая ее, вызвав метод `destroy`. Помимо единственного первичного ключа, метод `destroy` может принимать несколько первичных ключей, массив первичных ключей или [коллекцию](#) первичных ключей:

```
Flight::destroy(1);  
  
Flight::destroy(1, 2, 3);  
  
Flight::destroy([1, 2, 3]);  
  
Flight::destroy(collect([1, 2, 3]));
```

Если вы используете [мягкое удаление моделей](#), вы можете окончательно удалить модели с помощью метода `ForceDestroy`:

```
Flight::forceDestroy(1);
```

Метод `destroy` загружает каждую модель отдельно и вызывает для них метод `delete`, чтобы сработали события `deleting` и `deleted` должным образом для каждой модели.

## Удаление моделей через запрос

Конечно, вы можете создать запрос Eloquent для удаления всех моделей, соответствующих условиям запроса. В этом примере мы удалим все рейсы, помеченные как неактивные. Как и массовые обновления, массовые удаления не вызывают никаких событий модели для удаляемых моделей:

```
$deleted = Flight::where('active', 0)->delete();
```

События модели Eloquent `deleting`, и `deleted` при массовом удалении не будут инициированы для удаленных моделей. Это связано с тем, что модели фактически не извлекаются при выполнении оператора `delete`.

## Программное удаление

Помимо фактического удаления записей из БД, Eloquent может также выполнять «программно удалять» модели. При таком удалении, они фактически не удаляются из БД. Вместо этого для модели устанавливается атрибут `deleted_at`, указывающий дату и время, когда модель была «удалена». Чтобы включить программное удаление модели, добавьте в модель трейт `Illuminate\Database\Eloquent\SoftDeletes`:

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

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

Трейт `SoftDeletes` автоматически типизирует атрибут `deleted_at` к экземпляру `DateTime / Carbon`.

Вам также следует добавить столбец `deleted_at` в таблицу БД. [Построитель схемы](#) Laravel содержит метод для создания этого столбца:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Теперь, когда вы вызываете метод `delete` модели, в столбце `deleted_at` будут установлены текущие дата и время. Однако, запись в базе данных модели останется в таблице. При запросе модели, использующей программное удаление, программно удаленные модели будут автоматически исключены из всех результатов запроса.

Чтобы определить, был ли данный экземпляр модели программно удален, используйте метод `trashed`:

```
if ($flight->trashed()) {
    // ...
}
```

## Восстановление программно удаленных моделей

По желанию можно «восстановить» программно удаленную модель. Чтобы восстановить такую модель, используйте метод `restore` экземпляра модели. Метод `restore` установит в столбце `deleted_at` модели значение `null`:

```
$flight->restore();
```

Вы также можете использовать метод `restore` в запросе для восстановления нескольких моделей. Опять же, как и другие «массовые» операции, это не вызовет никаких событий модели для восстанавливаемых моделей:

```
App\Models\Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

Метод `restore` также используется при построении запросов, использующих [отношения](#):

```
$flight->history()->restore();
```

## Удаление моделей без возможности восстановления

По желанию можно действительно удалить модель из БД. Вы можете использовать метод `forceDelete`, чтобы окончательно удалить модель из таблицы БД:

```
$flight->forceDelete();
```

Вы также можете использовать метод `forceDelete` при построении запросов, использующих отношения Eloquent:

```
$flight->history()->forceDelete();
```

## Запросы для моделей, использующих программное удаление

## Включение программно удаленных моделей

Как отмечалось выше, программно удаленные модели будут автоматически исключены из результатов запроса. Однако, вы можете принудительно отобразить такие модели в результирующем наборе, используя метод `withTrashed` в запросе:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

Метод `withTrashed` также используется в запросе, использующем [отношения](#):

```
$flight->history()->withTrashed()->get();
```

## Извлечение только программно удаленных моделей

Метод `onlyTrashed` будет извлекать **только** программно удаленные модели:

```
$flights = Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

## # Периодическое удаление (pruning) старых записей

Иногда вам может понадобиться периодически удалять данные, которые больше не нужны. Для этого вы можете добавить трейт `Illuminate\Database\Eloquent\Prunable` или `Illuminate\Database\Eloquent\MassPrunable` к моделям, данные которых вы хотите периодически удалять. После добавления одного из этих признаков к модели, реализуйте метод `prunable`, который возвращает конструктор запросов Eloquent, разрешающий модели, которые больше не нужны:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Получите запрос для удаления устаревших записей модели.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Помечая модели как `Prunable`, вы также можете определить метод `pruning` для модели. Этот метод будет вызван перед удалением модели. Этот метод может быть полезен для удаления любых дополнительных ресурсов, связанных с моделью, таких как хранимые файлы, до того, как модель будет окончательно удалена из базы данных:

```
/*
 * Prepare the model for pruning.
 */
protected function pruning(): void
{
    // ...
}
```

После настройки модели с возможностью обрезки вы должны добавить в планировщик Artisan-команду `model:prune`, отредактировав файл `routes/console.php` вашего приложения. Вы можете выбрать подходящий интервал, через который будет выполняться эта команда:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('model:prune')->daily();
```

Команда `model:prune` автоматически обнаружит “обрезаемые” модели в каталоге `app/Models` вашего приложения. Если ваши модели находятся в другом месте, вы

можете использовать опцию `--model` для указания имен классов моделей:

```
Schedule::command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

Если вы хотите исключить обрезку определенных моделей при обрезке всех других обнаруженных моделей, вы можете использовать опцию `--except`:

```
Schedule::command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

Вы можете проверить свой `prunable` запрос, выполнив команду `model:prune` с опцией `--pretend`. При таком запуске команда `model:prune` просто сообщит, сколько записей было бы обрезано, если бы команда действительно выполнялась:

```
php artisan model:prune --pretend
```

Программно удалённые записи будут удалены из БД навсегда (`forceDelete`), если они попадают под критерий очистки.

## Очистка методами массового удаления

Когда модели помечены признаком `Illuminate\Database\Eloquent\MassPrunable`, модели удаляются из базы данных с помощью запросов массового удаления. Поэтому метод `pruning` не будет вызван, также не будут диспетчеризированы события `deleting` и `deleted` модели. Это происходит потому, что модели никогда не извлекаются перед удалением, что делает процесс обрезки гораздо более эффективным:

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

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

## # Репликация (тиражирование) моделей

Вы можете создать несохраненную копию существующего экземпляра модели, используя метод `replicate`. Этот метод особенно полезен, когда у вас есть экземпляры модели, которые имеют много одинаковых атрибутов:

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

Чтобы **исключить** один или несколько атрибутов из репликации в новую модель, вы можете передать массив в метод `replicate`:

```
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);

```

## # Диапазоны запроса (scopes)

### Глобальные диапазоны

Глобальные диапазоны позволяют добавлять ограничения ко всем запросам для конкретной модели. [Программное удаление](#) в Laravel использует глобальные диапазоны для получения только «не удаленных» моделей из БД. Написание пользовательских глобальных диапазонов предоставляют удобный и простой способ, гарантирующий, что в каждом запросе конкретной модели будут применены определенные ограничения.

### Генерация диапазонов

Чтобы сгенерировать новый глобальный диапазон, вы можете вызвать команду Artisan `make:scope`, которая поместит сгенерированный диапазон в каталог `app/Models/Scopes` вашего приложения:

```
php artisan make:scope AncientScope
```

### Написание глобальных диапазонов

Написание глобального диапазона – это просто. Сначала используйте команду `make:scope`, чтобы создать класс, реализующий интерфейс `Illuminate\Database\Eloquent\Scope`. Интерфейс `Scope` требует, чтобы вы реализовали один метод: `apply`. В методе `apply` можно добавлять условия `where` или другие типы ограничений к запросу по мере необходимости:

```
<?php
```

```
namespace App\Models\Scopes

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Применить диапазон к переданному построителю запросов.
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

Если диапазон добавляет столбцы в конструкцию Select-запроса, вы должны использовать метод `addSelect` вместо `select`. Это предотвратит непреднамеренную замену существующих Select-конструкций в запросе.

## Применение глобальных диапазонов

Чтобы назначить глобальный диапазон, вы можете просто добавить атрибут `ScopedBy` к модели.

```
<?php
```

```
namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Attributes\ScopedBy;

#[ScopedBy([AncientScope::class])]
class User extends Model
{
```

```
//  
}
```

Или же, вы можете вручную зарегистрировать глобальный диапазон, переопределив метод `booted` и вызвать метод модели `addGlobalScope`. Метод `addGlobalScope` принимает экземпляр вашего диапазона как единственный аргумент:

```
<?php  
  
use App\Models\Scopes\AncientScope;  
  
use App\Scopes\AncientScope;  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model  
{  
    /**  
     * Метод «booted» модели.  
     */  
    protected static function booted(): void  
    {  
        static::addGlobalScope(new AncientScope);  
    }  
}
```

После добавления диапазона в приведенном выше примере к модели `App\Models\User` вызов метода `User::all()` выполнит следующий SQL-запрос:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

## Анонимные глобальные диапазоны

Eloquent также позволяет вам определять глобальные диапазоны с использованием замыканий, что особенно полезно для простейших диапазонов, которые не требуют отдельного класса. При определении глобального диапазона с помощью замыкания вы должны указать имя диапазона по вашему выбору в качестве первого аргумента метода `addGlobalScope`:

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

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Метод «booted» модели.
     */
    protected static function booted(): void
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

## Игнорирование глобальных диапазонов

Для исключения глобального диапазона в текущем запросе, используйте метод `withoutGlobalScope`. Этот метод принимает имя класса глобального диапазона в качестве единственного аргумента:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Или, если вы определили глобальный диапазон с помощью замыкания, вы должны передать строковое имя, которое вы присвоили глобальному диапазону:

```
User::withoutGlobalScope('ancient')->get();
```

Если вы хотите удалить несколько или даже все глобальные диапазоны запроса, вы можете использовать метод `withoutGlobalScopes`:

```
// Игнорировать все глобальные диапазоны ...
User::withoutGlobalScopes()->get();

// Игнорировать некоторые глобальные диапазоны ...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

# Локальные диапазоны

Локальные диапазоны позволяют определять общие наборы ограничений запросов, которые можно легко повторно использовать в приложении. Например, вам может потребоваться часто получать всех пользователей, которые считаются «популярными». Чтобы определить диапазон, добавьте к методу модели Eloquent префикс `scope`.

Диапазоны должны всегда возвращать один и тот же экземпляр построителя запросов или `void`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Диапазон запроса, включающий только популярных пользователей.
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * Диапазон запроса, включающий только активных пользователей.
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

## Использование локальных диапазонов

После определения диапазона можно вызвать метод при выполнении запроса модели. Однако вы не должны включать префикс `scope` при вызове метода. Вы можете даже связывать вызовы с различными диапазонами:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Объединение нескольких диапазонов модели Eloquent с помощью оператора запроса `or` может потребовать использования замыканий для достижения правильной логической группировки:

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

Поскольку это может быть громоздким, то Eloquent содержит метод `orWhere` «более высокого порядка», который позволяет вам свободно связывать диапазоны без использования замыканий:

```
$users = User::popular()->orWhere->active()->get();
```

## Динамические диапазоны

По желанию можно определить диапазон, который принимает параметры. Для начала просто добавьте дополнительные параметры в сигнатуру метода диапазона. Параметры диапазона должны быть определены после параметра `$query`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Диапазон запроса, включающий пользователей только определенного типа.
     */
    public function scopeOfType(Builder $query, string $type): void
    {
        return $query->where('type', $type);
```

```
    }  
}
```

После того как ожидаемые аргументы были добавлены в сигнатуру метода диапазона, вы можете передать аргументы при вызове диапазона:

```
$users = User::ofType('admin')->get();
```

## # Сравнение моделей

По желанию можно определить, являются ли две модели «одинаковыми» или нет. Методы `is` и `isNot` могут использоваться для быстрой проверки наличия у двух моделей одного и того же первичного ключа, таблицы и соединения с базой данных:

```
if ($post->is($anotherPost)) {  
    // ...  
}  
  
if ($post->isNot($anotherPost)) {  
    // ...  
}
```

Методы `is` и `isNot` также доступны при использовании [отношений belongsTo, hasOne, morphTo, и morphOne](#). Этот метод особенно полезен, если вы хотите сравнить связанную модель без запроса на получение этой модели:

```
if ($post->author()->is($user)) {  
    // ...  
}
```

## # События

Хотите транслировать события Eloquent непосредственно в приложение на стороне

клиента? Ознакомьтесь с [model event broadcasting] (/docs/11.x/broadcasting#model-broadcasting) в Laravel.

Модели Eloquent инициируют некоторые события, что позволяет использовать следующие хуки жизненного цикла модели: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleting`, `forceDeleted`, `restoring`, `restored` и `replicating`.

Событие `retrieved` сработает, когда существующая модель будет извлечена из БД. Когда новая модель сохраняется в первый раз, инициируются события `creating` и `created`. События `updating` / `updated` будут инициированы при изменении существующей модели и вызове метода `save`. События `saving` / `saved` будут инициированы при создании или обновлении модели – даже если атрибуты модели не были изменены. События, заканчивающиеся на `-ing`, инициируются до сохранения изменений в модели, а события, заканчивающиеся на `-ed`, инициируются после сохранения изменений в модели.

Чтобы начать прослушивание событий модели, определите свойство `$dispatchesEvents` в модели Eloquent. Это свойство сопоставляет различные хуки жизненного цикла модели Eloquent с вашими собственными [классами событий](#). Каждый класс событий модели должен ожидать получения экземпляра затронутой модели через свой конструктор:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Карта событий для модели.
     *
     * @var array<string, string>
```

```
 */
protected $dispatchesEvents = [
    'saved' => UserSaved::class,
    'deleted' => UserDeleted::class,
];
}
```

После определения и сопоставления событий вы можете использовать [слушателей событий](#) для их обработки.

События модели Eloquent `saved`, `updated`, `deleting`, и `deleted` при массовом обновлении или удалении **не будут инициированы** для затронутых моделей. Это связано с тем, что модели фактически не извлекаются при массовом обновлении или удалении.

## Использование замыканий

Вместо использования пользовательских классов событий можно регистрировать замыкания, которые выполняются при иницировании различных событий модели. Как правило, вы должны зарегистрировать эти замыкания в методе `booted` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Метод «booted» модели.
     */
    protected static function booted(): void
    {
        static::created(function (User $user) {
            // ...
        });
    }
}
```

```
    }  
}
```

При необходимости вы можете использовать [поочередных анонимных слушателей событий](#) при регистрации событий модели. Это проинструктирует Laravel выполнить слушателя событий модели в фоновом режиме, используя [очередь](#) вашего приложения:

```
use function Illuminate\Events\queueable;  
  
static::created(queueable(function (User $user) {  
    // ...  
}));
```

## Наблюдатели

### Определение наблюдателей

Если прослушивается множество событий в модели, то можно использовать наблюдателей, чтобы сгруппировать пользовательских слушателей в одном классе. Классы наблюдателей имеют имена методов, созвучные событиям Eloquent, которые необходимо прослушивать. Каждый из этих методов получает затронутую модель в качестве единственного аргумента. Чтобы сгенерировать нового наблюдателя, используйте команду [make:observer Artisan](#):

```
php artisan make:observer UserObserver --model=User
```

Эта команда поместит новый класс наблюдателя в каталог [app/Observers](#) вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его. Созданный наблюдатель может выглядеть следующим образом:

```
<?php  
  
namespace App\Observers;  
  
use App\Models\User;  
  
class UserObserver
```

```

{
    /**
     * Обработать событие «created» модели User.
     */
    public function created(User $user): void
    {
        // ...
    }

    /**
     * Обработать событие «updated» модели User.
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
     * Обработать событие «deleted» модели User.
     */
    public function deleted(User $user): void
    {
        // ...
    }

    /**
     * Обработать событие «restored» модели User.
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * Обработать событие «forceDeleted» модели User.
     */
    public function forceDeleted(User $user): void
    {
        // ...
    }
}

```

Для регистрации наблюдателя вы можете просто добавить атрибут `ObservedBy` к соответствующей модели:

```

use App\Observers\UserObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

```

```
# [ObservedBy([UserObserver::class])]  
class User extends Authenticatable  
{  
    //  
}
```

Или же, вы можете вручную зарегистрировать наблюдателя, вызвав метод `observe` модели, которую вы хотите наблюдать. Вы можете зарегистрировать наблюдателей в методе `boot` класса `AppServiceProvider` вашего приложения:

```
use App\Models\User;  
use App\Observers\UserObserver;  
  
/**  
 * Запуск любых служб приложения.  
 */  
public function boot(): void  
{  
    User::observe(UserObserver::class);  
}
```

Существуют дополнительные события, которые может прослушивать наблюдатель, такие как `saving` и `retrieved`. Эти события описаны в документации [events](#).

## Наблюдатели и транзакции базы данных

Когда модели создаются в рамках транзакции базы данных, можно дать команду наблюдателю выполнять его обработчики событий только после того, как транзакция базы данных будет зафиксирована. Для этого вы можете реализовать интерфейс `ShouldHandleEventsAfterCommit` в вашем наблюдателе. При отсутствии транзакции, обработчики событий будут выполнены немедленно:

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

```
use App\Models\User;
use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;

class UserObserver
{
    /**
     * Обработать событие «created» модели User.
     */
    public function created(User $user): void
    {
        // ...
    }
}
```

## Подавление событий

По желанию можно временно «заглушить» все события, запускаемые моделью. Вы можете добиться этого, используя метод `withoutEvents`. Метод `withoutEvents` принимает замыкание как единственный аргумент. Любой код, выполняемый в этом замыкании, не будет запускать события модели и любое значение, возвращаемое переданным замыканием, будет возвращено методом `withoutEvents`:

```
use App\Models\User;

$user = User::withoutEvents(function () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

## Тихое сохранение одной модели

По желанию можно «сохранить» конкретную модель, не вызывая никаких событий. Вы можете сделать это с помощью метода `saveQuietly`:

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

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

Вы также можете "обновить", "удалить", "мягко удалить", "восстановить" и "реплицировать" заданную модель без отправки каких-либо событий:

```
$user->deleteQuietly();
$user->forceDeleteQuietly();
$user->restoreQuietly();
```

# Eloquent · Отношения

## # Введение

### # Определение отношений

- # Один к одному / Имеет один
- # Один ко многим / Имеет много
- # Модели по умолчанию
- # Один из многих
- # Один через отношение
- # Многие через отношение

### # Отношения Многие ко многим

- # Получение столбцов сводной таблицы
- # Фильтрация запросов по столбцам сводной таблицы
- # Сортировка запросов по столбцам сводной таблицы
- # Определение пользовательских моделей сводных таблиц

### # Полиморфные отношения

- # Один к одному (полиморфное)
- # Один ко многим (полиморфное)
- # Один из многих (полиморфное)
- # Многие ко многим (полиморфное)
- # Именование полиморфных типов

### # Динамические отношения

### # Запросы отношений

- # Методы отношений против динамических свойств
- # Запрос наличия отношений
- # Запрос отсутствия отношений
- # Запрос полиморфных отношений Morph To

### # Агрегирование связанных моделей

- # Подсчет связанных моделей
- # Другие агрегатные функции
- # Подсчет связанных моделей отношений Morph To

## # Жадная (eager) загрузка

- # Ограничение жадной загрузки
- # Жадная пост-загрузка
- # Предотвращение ленивой загрузки

## # Вставка и обновление связанных моделей

- # Метод Save
- # Метод Create
- # Обновление отношений Один К
- # Обновление отношений Многие ко многим

## # Затрагивание временных меток родителя

# # Введение

Таблицы базы данных часто связаны друг с другом. Например, пост в блоге может содержать много комментариев или заказ может быть связан с пользователем, который его разместил. Eloquent упрощает управление этими отношениями и работу с ними, а также поддерживает множество общих отношений:

- [Один к одному](#)
- [Один ко многим](#)
- [Многие ко многим](#)
- [Один через отношение](#)
- [Многие через отношение](#)
- [Один к одному \(полиморфное\)](#)
- [Один ко многим \(полиморфное\)](#)
- [Многие ко многим \(полиморфное\)](#)

# # Определение отношений

Отношения Eloquent определяются как методы в классах модели Eloquent.

Поскольку отношения реализованы поверх [построителей запросов](#), использование отношений как методов (к примеру, не `->posts`, а `->posts()`) обеспечивает

возможность создания цепочек методов и запросов. Например, мы можем добавить к результатам отношения дополнительное ограничение:

```
$user->posts()->where('active', 1)->get();
```

Но, прежде чем углубляться в использование отношений, давайте узнаем, как определить каждый тип отношений, поддерживаемый Eloquent.

## Один к одному / Имеет один

Отношения «один-к-одному» – это очень простой тип отношений базы данных. Например, модель `User` может быть связана с одной моделью `Phone`. Чтобы определить это отношение, мы поместим метод `phone` в модель `User`. Метод `phone` должен вызывать метод `hasOne` и возвращать его результат. Метод `hasOne` доступен для вашей модели через базовый класс `Illuminate\Database\Eloquent\Model` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Получить телефон, связанный с пользователем.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

Первым аргументом, передаваемым методу `hasOne`, является имя связанного класса модели. Как только связь определена, мы можем получить связанную запись, используя динамические свойства Eloquent. Динамические свойства позволяют получить доступ к методам отношений, как если бы они были свойствами, определенными в модели:

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

Eloquent определяет внешний ключ отношения на основе имени родительской модели. В этом случае автоматически предполагается, что модель `Phone` имеет внешний ключ `user_id`. Если вы хотите переопределить это соглашение, вы можете передать второй аргумент методу `hasOne`:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

Кроме того, Eloquent предполагает, что внешний ключ должен иметь значение, соответствующее столбцу первичного ключа родительского элемента. Другими словами, Eloquent будет искать значение столбца `id` пользователя в столбце `user_id` модели `Phone`. Если вы хотите, чтобы отношение использовало значение первичного ключа, отличное от `id` или свойства вашей модели `$primaryKey`, вы можете передать третий аргумент методу `hasOne`:

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

## Определение обратной связи Один к одному

Итак, мы можем получить доступ к модели `Phone` из нашей модели `User`. Затем давайте определим отношение в модели `Phone`, позволяющее нам получить доступ к пользователю, которому принадлежит телефон. Мы можем определить инверсию отношения `hasOne` с помощью метода `belongsTo`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Получить пользователя, владеющего телефоном.
     */
    public function user(): BelongsTo
    {
```

```
        return $this->belongsTo(User::class);
    }
}
```

При вызове метода `user`, Eloquent попытается найти модель `User`, у которой есть `id`, который соответствует столбцу `user_id` в модели `Phone`.

Eloquent определяет имя внешнего ключа, анализируя имя метода отношения и добавляя к имени метода суффикс `_id`. Итак, в этом случае Eloquent предполагает, что модель `Phone` имеет столбец `user_id`. Однако, если внешний ключ в модели `Phone` не является `user_id`, вы можете передать собственное имя ключа в качестве второго аргумента методу `belongsTo`:

```
/**
 * Получить пользователя, владеющего телефоном.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

Если родительская модель не использует `id` в качестве первичного ключа или вы хотите найти связанную модель, используя другой столбец, вы можете передать третий аргумент методу `belongsTo`, указав ваш ключ родительской таблицы:

```
/**
 * Получить пользователя, владеющего телефоном.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

## ОДИН КО МНОГИМ / ИМЕЕТ МНОГО

Отношение «один-ко-многим» используется для определения отношений, в которых одна модель является родительской для одной или нескольких дочерних моделей. Например, пост в блоге может содержать бесконечное количество комментариев. Как и все другие отношения Eloquent, отношения «один-ко-многим» определяются путем определения метода в вашей модели Eloquent:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    /**
     * Получить комментарии к посту блога.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

Помните, что Eloquent автоматически определит правильный столбец внешнего ключа для модели `Comment`. По соглашению Eloquent берет имя родительской модели в «змеином регистре» и добавляет к нему суффикс `_id`. Итак, в этом примере Eloquent предполагает, что столбец внешнего ключа в модели `Comment` именуется `post_id`.

Как только метод отношения определен, мы можем получить доступ к [коллекции](#) связанных комментариев, используя свойство `comments`. Поскольку Eloquent обеспечивает «динамические свойства отношений», то мы можем получить доступ к методам отношений, как если бы они были определены как свойства в модели:

```
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    // ...
}
```

Поскольку все отношения построены на базе построителей запросов, вы можете добавить дополнительные ограничения в запрос отношения, вызвав метод `comments` и продолжая связывать условия с запросом:

```
$comment = Post::find(1)->comments()
    ->where('title', 'foo')
    ->first();
```

Подобно методу `hasOne`, вы также можете переопределить внешние и локальные ключи, передав дополнительные аргументы методу `hasMany`:

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

## Автоматическая гидратация родительских моделей на дочерних объектах

Даже при использовании быстрой загрузки Eloquent могут возникнуть проблемы с запросами «N + 1», если вы попытаетесь получить доступ к родительской модели из дочерней модели при циклическом переборе дочерних моделей:

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

foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->post->title;
    }
}
```

В приведенном выше примере возникла проблема запроса «N + 1», поскольку, хотя комментарии были готовы загружаться для каждой модели `Post`, Eloquent не выполняет автоматическую гидратацию родительской `Post` в каждой дочерней модели `Comment`.

Если вы хотите, чтобы Eloquent автоматически переносил родительские модели в свои дочерние элементы, вы можете вызвать метод `chaperone` при определении отношения `hasMany`:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class)->chaperone();
    }
}
```

Или, если вы хотите включить автоматическую гидратацию родительского элемента во время выполнения, вы можете вызвать модель `chaperone` при загрузке связи:

```
use App\Models\Post;

$post = Post::with([
    'comments' => fn ($comments) => $comments->chaperone(),
])->get();
```

## Определение обратной связи ОДИН КО МНОГИМ

Теперь, когда мы можем получить доступ ко всем комментариям поста, давайте определим отношение, чтобы разрешить комментарию доступ к его родительскому посту. Чтобы определить инверсию отношения `hasMany`, определите метод отношения в дочерней модели, который вызывает метод `belongsTo`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * Получить пост, которому принадлежит комментарий.
     */
    public function post(): BelongsTo
```

```
{  
    return $this->belongsTo(Post::class);  
}  
}
```

Как только связь определена, мы можем получить родительский пост комментария, обратившись к «динамическому свойству отношения» `post`:

```
use App\Models\Comment;  
  
$comment = Comment::find(1);  
  
return $comment->post->title;
```

В приведенном выше примере Eloquent попытается найти модель `Post`, у которой есть `id`, который соответствует столбцу `post_id` в модели `Comment`.

Eloquent определяет имя внешнего ключа по умолчанию, анализируя имя метода отношения и добавляя к имени метода суффикс `_`, за которым следует имя столбца первичного ключа родительской модели. Итак, в этом примере Eloquent предполагает, что внешний ключ модели `Post` в таблице `comments` – это `post_id`.

Однако, если внешний ключ для ваших отношений не соответствует этим соглашениям, вы можете передать свое имя внешнего ключа в качестве второго аргумента методу `belongsTo`:

```
/**  
 * Получить пост, которому принадлежит комментарий.  
 */  
public function post(): BelongsTo  
{  
    return $this->belongsTo(Post::class, 'foreign_key');  
}
```

Если ваша родительская модель не использует `id` в качестве первичного ключа или вы хотите найти связанную модель, используя другой столбец, то вы можете передать третий аргумент методу `belongsTo`, указав свой ключ родительской таблицы:

```
/**  
 * Получить пост, которому принадлежит комментарий.  
 */  
public function post(): BelongsTo  
{  
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');  
}
```

## Модели по умолчанию

Отношения `belongsTo`, `hasOne`, `hasOneThrough` и `morphOne` позволяют вам определить модель по умолчанию, которая будет возвращена, если данное отношение равно `null`. Этот шаблон часто называют [шаблоном нулевого объекта](#), который поможет удалить условные проверки в вашем коде. В следующем примере отношение `user` вернет пустую модель `App\Models\User`, если к модели `Post` не привязан ни один `user`:

```
/**  
 * Получить автора поста.  
 */  
public function user(): BelongsTo  
{  
    return $this->belongsTo(User::class)->withDefault();  
}
```

Чтобы заполнить модель по умолчанию атрибутами, вы можете передать массив или замыкание методу `withDefault`:

```
/**  
 * Получить автора поста.  
 */  
public function user(): BelongsTo  
{  
    return $this->belongsTo(User::class)->withDefault([  
        'name' => 'Guest Author',  
    ]);  
}  
  
/**  
 * Получить автора поста.  
 */  
public function user(): BelongsTo  
{  
    return $this->belongsTo(User::class)->withDefault(function ($user, $post) {
```

```
$user->name = 'Guest Author';  
});  
}
```

## Запрос отношений Один К

При запросе дочерних элементов отношения “принадлежит” (`belongs to`) вы можете вручную создать предложение `where` для получения соответствующих моделей Eloquent:

```
use App\Models\Post;  
  
$posts = Post::where('user_id', $user->id)->get();
```

Однако вам может быть удобнее использовать метод `whereBelongsTo`, который автоматически определит правильные отношения и внешний ключ для данной модели:

```
$posts = Post::whereBelongsTo($user)->get();
```

Вы также можете предоставить экземпляр коллекции методу `whereBelongsTo`. При этом Laravel будет извлекать модели, принадлежащие любой из родительских моделей внутри коллекции:

```
$users = User::where('vip', true)->get();  
  
$posts = Post::whereBelongsTo($users)->get();
```

По умолчанию Laravel будет определять отношения, связанные с данной моделью, на основе имени класса модели; однако вы можете указать имя отношения вручную, указав его в качестве второго аргумента метода `whereBelongsTo`:

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

## ОДИН ИЗ МНОГИХ

Иногда модель может иметь множество связанных моделей, но вы хотите легко получить “самую последнюю” или “самую старую” связанную модель. Например, модель `User` может быть связана со многими моделями `Order`, но вы хотите определить удобный способ взаимодействия с последним заказом пользователя. Для этого можно использовать тип отношения `hasOne` в сочетании с методами `ofMany`:

```
/**  
 * Получить последний (самый новый) заказ пользователя  
 */  
public function latestOrder(): HasOne  
{  
    return $this->hasOne(Order::class)->latestOfMany();  
}
```

Аналогично, вы можете определить метод для получения “самой старой” (первой) связанной модели отношения:

```
/**  
 * Получить самый старый заказ пользователя  
 */  
public function oldestOrder(): HasOne  
{  
    return $this->hasOne(Order::class)->oldestOfMany();  
}
```

По умолчанию методы `latestOfMany` и `oldestOfMany` извлекают самую последнюю или самую старую связанную модель на основе *первичного ключа* модели. Этот ключ должен быть сортируемым. Чтобы использовать данные другого столбца, используйте метод `ofMany`. Он принимает в качестве первого аргумента сортируемый столбец и то, какую агрегатную функцию (`min` или `max`) применить при запросе связанной модели.

Например, так можно получить самый дорогой заказ пользователя:

```
/**  
 * Получить самый дорогой заказ пользователя  
 */  
public function largestOrder(): HasOne  
{
```

```
    return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

Поскольку PostgreSQL не поддерживает выполнение функции `MAX` для столбцов UUID, в настоящее время невозможно использовать отношения “один-из-многих” в сочетании со столбцами UUID PostgreSQL.

## Преобразование отношений "Many" в отношения "Has One"

Часто, при получении одной модели с использованием методов `latestOfMany`, `oldestOfMany` или `ofMany`, у вас уже определены отношения “has many” для той же модели. Для удобства Laravel позволяет легко преобразовать это отношение в отношение “has one”, вызвав метод `one` отношения:

```
/**
 * Получить заказы пользователя
 */
public function orders(): HasMany
{
    return $this->hasMany(Order::class);
}

/**
 * Получить самый дорогой заказ пользователя
 */
public function largestOrder(): HasOne
{
    return $this->orders()->one()->ofMany('price', 'max');
}
```

## Продвинутые возможности отношения Один-из-многих

Критерии сортировки при выборе могут быть сложными. Например, модель `Product` может иметь множество связанных с ней моделей `Price`, которые сохраняются в системе даже после публикации новых цен – например, чтобы можно было посмотреть динамику изменения цены продукта. Кроме того, новые данные о

ценах на продукт могут быть опубликованы заранее, чтобы вступить в силу в определённую дату – через колонку `published_at`.

Таким образом, нам нужно получить последние опубликованные цены, если дата публикации не находится в будущем. Кроме того, если две цены имеют одинаковую дату публикации, мы предпочтем цену с наибольшим ID. Для этого в метод `ofMany` нужно передать массив, который содержит сортируемые столбцы, определяющие последнюю цену. Кроме того, в качестве второго аргумента метода `ofMany` передаётся функция, которая будет отвечать за добавление дополнительного фильтра по дате публикации:

```
/**  
 * Получить актуальную цену на продукт  
 */  
public function currentPricing(): HasOne  
{  
    return $this->hasOne(Price::class)->ofMany([  
        'published_at' => 'max',  
        'id' => 'max',  
    ], function (Builder $query) {  
        $query->where('published_at', '<', now());  
    });  
}
```

## Один через отношение

Отношение «один-через-отношение» определяет отношение «один-к-одному» с другой моделью. Однако, это отношение указывает на то, что декларируемую модель можно сопоставить с одним экземпляром другой модели, связавшись через третью модель.

Например, в приложении автомастерской каждая модель `Mechanic` может быть связана с одной моделью `Car`, и каждая модель `Car` может быть связана с одной моделью `Owner`. В то время как механик и владелец не имеют прямых отношений в базе данных, механик может получить доступ к владельцу через модель `Car`. Давайте посмотрим на таблицы, необходимые для определения этой связи:

```
mechanics  
  id - integer  
  name - string  
  
cars
```

```
id - integer
model - string
mechanic_id - integer

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

Теперь, когда мы изучили структуру таблицы для отношения, давайте определим отношения в модели [Mechanic](#):

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOneThrough;

class Mechanic extends Model
{
    /**
     * Получить владельца машины.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

Первый аргумент, передаваемый методу [hasOneThrough](#) – это имя последней модели, к которой мы хотим получить доступ, а второй аргумент – это имя промежуточной (сводной) модели.

Или, если соответствующие отношения уже определены для всех моделей, участвующих в отношении, вы можете легко определить отношение “один-через-отношение”, вызвав метод [through](#) и указав имена этих отношений. Например, если у модели [Mechanic](#) есть отношение [cars](#), а у модели [Car](#) есть отношение [owner](#), вы можете определить отношение “один-через-отношение”, соединяющее механика и владельца, следующим образом:

```
// String based syntax...
return $this->through('cars')->has('owner');
```

```
// Dynamic syntax...
return $this->throughCars()->hasOwner();
```

## Соглашения по именованию ключей

### отношения Один через отношение

Типичные соглашения о внешнем ключе Eloquent будут использоваться при выполнении запросов отношения. Если вы хотите изменить ключи отношения, вы можете передать их в качестве третьего и четвертого аргументов методу `hasOneThrough`. Третий аргумент – это имя внешнего ключа сводной модели. Четвертый аргумент – это имя внешнего ключа окончательной модели. Пятый аргумент – это локальный ключ, а шестой аргумент – это локальный ключ сводной модели:

```
class Mechanic extends Model
{
    /**
     * Получить владельца машины.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Внешний ключ в таблице `cars` ...
            'car_id', // Внешний ключ в таблице `owners` ...
            'id', // Локальный ключ в таблице `mechanics` ...
            'id' // Локальный ключ в таблице `cars` ...
        );
    }
}
```

Или, как обсуждалось ранее, если соответствующие отношения уже определены для всех моделей, участвующих в отношении, вы можете легко определить отношение “один-через-отношение”, вызвав метод `through` и указав имена этих отношений. Этот подход предоставляет преимущество повторного использования соглашений по ключам, уже определенных в существующих отношениях:

```
// String based syntax...
return $this->through('cars')->has('owner');
```

```
// Dynamic syntax...
return $this->throughCars()->hasOwner();
```

## Многие через отношение

Отношение «многие-через-отношение» обеспечивает удобный способ доступа к отдаленным отношениям через промежуточное отношение. Например, предположим, что мы создаем платформу развертывания, такую как [Laravel Vapor](#). Модель `Project` может получить доступ ко многим моделям `Deployment` через сводную модель `Environment`. Используя этот пример, вы можете легко собрать все развертывания (deployments) для конкретного проекта. Давайте посмотрим на таблицы, необходимые для определения этой связи:

```
projects
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

Теперь, когда мы изучили структуру таблицы для отношения, давайте определим отношение в модели `Project`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;

class Project extends Model
{
    /**
     * Получить все развертывания для проекта.
     */
    public function deployments(): HasManyThrough
```

```
{  
    return $this->hasManyThrough(Deployment::class, Environment::class);  
}  
}
```

Первый аргумент, передаваемый методу `hasManyThrough` – это имя последней модели, к которой мы хотим получить доступ, а второй аргумент – это имя сводной модели.

Или, если соответствующие отношения уже определены для всех моделей, участвующих в отношении, вы можете легко определить отношение «многие-через-отношение», вызвав метод `through` и указав имена этих отношений.

Например, если у модели `Project` есть отношение `environments`, а у модели `Environment` есть отношение `deployments`, вы можете определить отношение «многие-через-отношение», соединяющее проект и деплойменты, следующим образом:

```
// String based syntax...  
return $this->through('environments')->has('deployments');  
  
// Dynamic syntax...  
return $this->throughEnvironments()->hasDeployments();
```

Хотя таблица модели `Deployment` не содержит столбца `project_id`, отношение `hasManyThrough` обеспечивает доступ к `deployments` проекта через `$project->deployments`. Чтобы получить эти модели, Eloquent проверяет столбец `project_id` в сводной таблице модели `Environment`. После нахождения соответствующих идентификаторов `environments` они используются для запроса таблицы модели `Deployment`.

## Соглашения по именованию ключей

### отношения Многие через отношение

Типичные соглашения о внешнем ключе Eloquent будут использоваться при выполнении запросов отношения. Если вы хотите изменить ключи отношения, вы можете передать их в качестве третьего и четвертого аргументов методу `hasManyThrough`. Третий аргумент – это имя внешнего ключа сводной модели. Четвертый аргумент – это имя внешнего ключа окончательной модели. Пятый аргумент – это локальный ключ, а шестой аргумент – это локальный ключ сводной модели:

```

class Project extends Model
{
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Внешний ключ в таблице `environments` ...
            'environment_id', // Внешний ключ в таблице `deployments` ...
            'id', // Локальный ключ в таблице `projects` ...
            'id' // Локальный ключ в таблице `environments` ...
        );
    }
}

```

Или, как было обсуждено ранее, если соответствующие отношения уже определены для всех моделей, участвующих в отношении, вы можете легко определить отношение «многие-через-отношение», вызвав метод `through` и указав имена этих отношений. Этот подход предоставляет преимущество повторного использования соглашений по ключам, уже определенных в существующих отношениях:

```

// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();

```

## # Отношения Многие ко многим

Отношения «многие-ко-многим» немного сложнее, чем отношения `hasOne` и `hasMany`. Примером отношения «многие-ко-многим» является пользователь, у которого много ролей, и эти роли также используются другими пользователями в приложении. Например, пользователю могут быть назначены роли «Автор» и «Редактор»; однако эти роли также могут быть назначены другим пользователям. Итак, у пользователя много ролей, а у роли много пользователей.

### Структура таблицы Многие ко многим

Чтобы определить эту связь, необходимы три таблицы базы данных: `users`, `roles`, и `role_user`. Таблица `role_user` является производной от имен связанных моделей в алфавитном порядке и содержит столбцы `user_id` и `role_id`. Эта таблица используется как промежуточная таблица, связывающая пользователей и роли.

Помните, поскольку роль может принадлежать многим пользователям, мы не можем просто разместить столбец `user_id` в таблице `role`. Это означало бы, что роль могла принадлежать только одному пользователю. Для обеспечения поддержки ролей, назначаемых нескольким пользователям, необходима таблица `role_user`. Мы можем резюмировать структуру таблицы отношений следующим образом:

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

## Структура модели Многие ко многим

Отношения «многие-ко-многим» определяются путем написания метода, который возвращает результат метода `belongsToMany`. Метод `belongsToMany` обеспечен базовым классом `Illuminate\Database\Eloquent\Model`, который используется всеми моделями Eloquent вашего приложения. Например, давайте определим метод `roles` в нашей модели `User`. Первым аргументом, передаваемым этому методу, является имя класса сводной модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
```

```
/**  
 * Роли, принадлежащие пользователю.  
 */  
public function roles(): BelongsToMany  
{  
    return $this->belongsToMany(Role::class);  
}  
}
```

Как только связь определена, вы можете получить доступ к ролям пользователя, используя динамическое свойство связи `roles`:

```
use App\Models\User;  
  
$user = User::find(1);  
  
foreach ($user->roles as $role) {  
    // ...  
}
```

Поскольку все отношения также служат в качестве построителей запросов, вы можете добавить дополнительные ограничения к запросу отношений, вызвав метод `roles` и продолжив связывать условия с запросом:

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

Чтобы определить имя промежуточной таблицы отношения, Eloquent соединит имена двух связанных моделей в алфавитном порядке. Однако вы можете изменить это соглашение, передав название таблицы в виде второго аргумента метода `belongsToMany`:

```
return $this->belongsToMany(Role::class, 'role_user');
```

В дополнение к переопределению имени промежуточной таблицы, вы также можете изменить имена столбцов ключей в таблице, передав дополнительные аргументы методу `belongsToMany`. Третий аргумент – это имя внешнего ключа модели, для которой вы определяете отношение. Четвертый аргумент – это имя внешнего ключа модели, к которой вы присоединяетесь:

```
return $this->hasMany(Role::class, 'role_user', 'user_id', 'role_id');
```

## Определение обратной связи Многие ко многим

Чтобы определить «обратное» отношение «многие-ко-многим», вы должны определить метод в связанной модели, который также возвращает результат метода `belongsToMany`. Чтобы завершить наш пример пользователи / роли, давайте определим метод `users` в модели `Role`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * Пользователи, принадлежащие к роли.
     */
    public function users(): BelongsToMany
    {
        return $this->hasMany(User::class);
    }
}
```

Как видите, отношение определяется точно так же, как и его аналог в модели `User`, за исключением ссылки на модель `App\Models\User`. Поскольку мы повторно используем метод `belongsToMany`, все стандартные параметры настройки таблиц и ключей доступны при определении «обратных» отношений «многие-ко-многим».

## Получение столбцов сводной таблицы

Как вы уже узнали, для работы с отношениями «многие-ко-многим» требуется наличие промежуточной таблицы. Eloquent предлагает несколько очень полезных способов взаимодействия с этой таблицей. Например, предположим, что наша модель `User` имеет много моделей `Role`, с которыми она связана. После доступа к этой связи мы можем получить доступ к промежуточной таблице с помощью атрибута `pivot` в моделях:

```
use App\Models\User;

$user = User::find(1);

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

Обратите внимание, что каждой модели `Role`, которую мы получаем, автоматически назначается атрибут `pivot`. Этот атрибут содержит модель, представляющую промежуточную таблицу.

По умолчанию в модели `pivot` будут присутствовать только ключи модели. Если ваша промежуточная таблица содержит дополнительные атрибуты, вы должны указать их при определении отношения:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

Если вы хотите, чтобы ваша промежуточная таблица имела временные метки `created_at` и `updated_at`, которые автоматически поддерживаются Eloquent, вызовите метод `withTimestamps` при определении отношения:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

Промежуточные таблицы, использующие автоматически поддерживаемые временные метки Eloquent, должны иметь столбцы временных меток `created_at` и `updated_at`.

## Корректировка имени атрибута `pivot`

Как отмечалось ранее, атрибуты из промежуточной таблицы могут быть доступны в моделях через атрибут `pivot`. Однако, вы можете изменить имя этого атрибута, чтобы лучше отразить его назначение в вашем приложении.

Например, если ваше приложение содержит пользователей, которые могут подписаться на подкасты, вы, вероятно, имеете отношение «многие-ко-многим» между пользователями и подкастами. По желанию можно переименовать атрибут `pivot` промежуточной таблицы на `subscription`. Это можно сделать с помощью метода `as` при определении отношения:

```
return $this->belongsToMany(Podcast::class)
    ->as('subscription')
    ->withTimestamps();
```

После указания атрибута промежуточной таблицы, вы можете получить доступ к данным промежуточной таблицы, используя указанное имя:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

## Фильтрация запросов по столбцам сводной таблицы

Вы также можете отфильтровать результаты, возвращаемые запросами отношения `belongsToMany`, используя методы `wherePivot`, `wherePivotIn`, `wherePivotNotIn`, `wherePivotBetween`, `wherePivotNotBetween`, `wherePivotNull` и `wherePivotNotNull` при определении отношения:

```
return $this->belongsToMany(Role::class)
    ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
    ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
    ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31 23:59:59']);

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31 23:59:59']);
```

```
return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
    ->as('subscriptions')
    ->wherePivotNotNull('expired_at');
```

## Сортировка запросов по столбцам сводной таблицы

Вы можете упорядочить результаты запросов отношений `belongsToMany`, используя метод `orderByPivot`. В следующем примере мы получим все последние значки для пользователя:

```
return $this->belongsToMany(Badge::class)
    ->where('rank', 'gold')
    ->orderByPivot('created_at', 'desc');
```

## Определение пользовательских моделей сводных таблиц

Если вы хотите определить собственную модель промежуточной таблицы отношения «многие-ко-многим», то вы можете вызвать метод `using` при определении отношения. Явные сводные модели дают вам возможность определять дополнительное поведение в модели сводной таблицы, такое как методы и приведения типов.

Явные сводные модели отношения «многие-ко-многим» должны расширять класс `Illuminate\Database\Eloquent\Relations\Pivot`, в то время как явные полиморфные сводные модели отношения «многие-ко-многим» должны расширять класс `Illuminate\Database\Eloquent\Relations\MorphPivot`. Например, мы можем определить модель `Role`, которая использует явную сводную модель `RoleUser`:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
```

```
class Role extends Model
{
    /**
     * Пользователи, принадлежащие к роли.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

При определении модели `RoleUser` вы должны расширять класс `Illuminate\Database\Eloquent\Relations\Pivot`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    // ...
}
```

Сводные модели не могут использовать трейт `SoftDeletes`. Если вам нужно программно удалить сводные записи, подумайте о преобразовании вашей сводной модели в реальную модель Eloquent.

## Пользовательские сводные модели и автоинкрементные идентификаторы

Если вы определили отношение «многие-ко-многим», которое использует явную сводную модель, и эта сводная модель имеет автоинкрементный первичный ключ, то вы должны убедиться, что ваш класс явной сводной модели определяет свойство `$incrementing`, для которого установлено значение `true`.

```
/**  
 * Указывает, что идентификаторы модели являются автоинкрементными.  
 *  
 * @var bool  
 */  
public $incrementing = true;
```

## # Полиморфные отношения

Полиморфные отношения позволяют дочерней модели принадлежать более чем к одному типу модели с использованием одной ассоциации. Например, представьте, что вы создаете приложение, которое позволяет пользователям делиться постами и видео в блогах. В таком приложении модель [Comment](#) может принадлежать как к моделям [Post](#), так и [Video](#).

### Один к одному (полиморфное)

#### Структура таблицы отношения

#### Один к одному (полиморфное)

Полиморфное отношение «один-к-одному» похоже на типичное «один-к-одному» отношение; однако, дочерняя модель может принадлежать более чем к одному типу моделей с помощью одной ассоциации. Например, блог [Post](#) и [User](#) могут иметь полиморфное отношение С моделью [Image](#). Использование полиморфного «один-к-одному» отношения позволяет вам иметь единую таблицу уникальных изображений, которые могут быть связаны с постами и пользователями. Сначала рассмотрим структуру таблицы:

```
posts  
id - integer  
name - string
```

```
users  
id - integer  
name - string
```

```
images  
id - integer  
url - string
```

```
imageable_id - integer
imageable_type - string
```

Обратите внимание на столбцы `imageable_id` и `imageable_type` в таблице `images`.

Столбец `imageable_id` будет содержать значение идентификатора поста или пользователя, а столбец `imageable_type` будет содержать имя класса родительской модели. Столбец `imageable_type` используется Eloquent для определения того, какой «типа» родительской модели возвращать при доступе к отношению `imageable`. В этом случае столбец будет содержать либо `App\Models\Post`, либо `App\Models\User`.

## Структура модели отношения

### Один к одному (полиморфное)

Давайте рассмотрим определения модели, необходимые для построения этой связи:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Image extends Model
{
    /**
     * Получить родительскую модель (пользователя или поста), к которой относится изображение.
     */
    public function imageable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class Post extends Model
{
    /**
     * Получить изображение поста.
     */
    public function image(): MorphOne
    {
```

```
        return $this->morphOne(Image::class, 'imageable');
    }

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;
class User extends Model
{
    /**
     * Получить изображение пользователя.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

## Получение отношения Один к одному (полиморфное)

Как только ваша таблица базы данных и модели определены, вы можете получить доступ к отношениям через свои модели. Например, чтобы получить изображение для поста, мы можем обратиться к динамическому свойству связи `image`:

```
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

Вы можете получить родительский объект полиморфной модели, обратившись к имени метода, который выполняет вызов `morphTo`. В данном случае это метод `imageable` модели `Image`. Итак, мы будем обращаться к этому методу как к динамическому свойству отношения:

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

Отношение `imageable` в модели `Image` будет возвращать экземпляр `Post` или `User`, в зависимости от того, к какому типу модели относится изображение.

## Соглашения по именованию ключей

### отношения Один к одному (полиморфное)

Если необходимо, то вы можете указать имя столбцов `id` и `type`, используемых вашей полиморфной дочерней моделью. Если вы это сделаете, то убедитесь, что вы всегда передаете имя отношения в качестве первого аргумента методу `morphTo`. Обычно это значение должно совпадать с именем метода, поэтому вы можете использовать константу `__FUNCTION__` PHP:

```
/**
 * Получить родительскую модель, к которой относится изображение.
 */
public function imageable(): MorphTo
{
    return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

## Один ко многим (полиморфное)

### Структура таблицы отношения

### Один ко многим (полиморфное)

Полиморфное отношение «один-ко-многим» похоже на типичное отношение «один-ко-многим»; однако, дочерняя модель может принадлежать более чем к одному типу моделей с помощью одной ассоциации. Например, представьте, что пользователи вашего приложения могут «комментировать» посты и видео. Используя полиморфные отношения, вы можете использовать одну таблицу `comments`, чтобы хранить комментарии как для постов, так и для видео. Во-первых, давайте рассмотрим структуру таблицы, необходимую для построения этой связи:

```
posts
  id - integer
  title - string
  body - text
```

```
videos
```

```
id - integer
title - string
url - string

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

## Структура модели отношения

### Один ко многим (полиморфное)

Давайте рассмотрим определения модели, необходимые для построения этой связи:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Comment extends Model
{
    /**
     * Получить родительскую модель (поста или видео), к которой относится комментарий
     */
    public function commentable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Post extends Model
{
    /**
     * Получить все комментарии поста.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

```
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Video extends Model
{
    /**
     * Получить все комментарии видео.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

## Получение отношения Один ко многим (полиморфное)

После того как ваша таблица базы данных и модели определены, вы можете получить доступ к отношениям через динамические свойства отношений вашей модели. Например, чтобы получить доступ ко всем комментариям к постам, мы можем использовать динамическое свойство `comments`:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    // ...
}
```

Вы также можете получить родительскую модель полиморфной дочерней модели, обратившись к имени метода, который выполняет вызов `morphTo`. В данном случае это метод `commentable` в модели `Comment`. Итак, мы будем обращаться к этому методу как к динамическому свойству связи, чтобы получить доступ к родительской модели комментария:

```
use App\Models\Comment;

$comment = Comment::find(1);
```

```
$commentable = $comment->commentable;
```

Отношение `commentable` в модели `Comment` вернет либо экземпляр `Post`, либо `Video`, в зависимости от того, какой тип модели является родительским для комментария.

## Автоматическое увлажнение родительских моделей на дочерних объектах

Даже при использовании быстрой загрузки Eloquent могут возникнуть проблемы с запросами «N + 1», если вы попытаетесь получить доступ к родительской модели из дочерней модели при циклическом переборе дочерних моделей:

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

foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->commentable->title;
    }
}
```

В приведенном выше примере возникла проблема запроса «N + 1», поскольку, хотя комментарии были готовы загружаться для каждой модели `Post`, Eloquent не выполняет автоматическую гидратацию родительской `Post` в каждой дочерней модели `Comment`.

Если вы хотите, чтобы Eloquent автоматически переносил родительские модели в свои дочерние элементы, вы можете вызвать метод `chaperone` при определении отношения `morphMany`:

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

Или, если вы хотите включить автоматическую гидратацию родительского элемента во время выполнения, вы можете вызвать модель `chaperone` при загрузке связи:

```
use App\Models\Post;

$posts = Post::with([
    'comments' => fn ($comments) => $comments->chaperone(),
])->get();
```

## Один из многих (полиморфное)

Иногда модель может иметь множество связанных моделей, но вы хотите легко получить “самую последнюю” или “самую старую” связанную модель. Например, модель `User` может быть связана со многими моделями `Image`, но вы хотите определить удобный способ взаимодействия с последним загруженным пользователем изображением. Для этого можно использовать тип отношения `morphOne` в сочетании с методами `ofMany`:

```
/**
 * Get the user's most recent image.
 */
public function latestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

Аналогично вы можете определить метод для получения “самой старой” (первой) связанной модели отношения:

```
/**
 * Get the user's oldest image.
 */
public function oldestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

По умолчанию методы `latestOfMany` и `oldestOfMany` извлекают самую последнюю или самую старую связанную модель на основе *первичного ключа* модели. Этот ключ

должен быть сортируемым. Чтобы использовать данные другого столбца, используйте метод `ofMany`. Он принимает в качестве первого аргумента сортируемый столбец и то, какую агрегатную функцию (`min` или `max`) применить при запросе связанной модели.

```
/**  
 * Получаем изображение, у которого больше всего лайков.  
 */  
public function bestImage()  
{  
    return $this->morphOne(Image::class, 'imageable')->ofMany('likes', 'max');  
}
```

Можно построить более сложные отношения “один из многих”. Для получения дополнительной информации обратитесь к [данному разделу документации].(#advanced-has-one-of-many-relationships).

## Многие ко многим (полиморфное)

### Структура таблицы отношения

#### Многие ко многим (полиморфное)

Полиморфные отношения «многие-ко-многим» немного сложнее, чем полиморфные отношения «один-к-одному» и «один-ко-многим». Например, модель `Post` и модель `Video` могут иметь полиморфное отношение к модели `Tag`. Использование полиморфного отношения «многие-ко-многим» в этой ситуации позволит вашему приложению иметь единую таблицу уникальных тегов, которые могут быть связаны с постами или видео. Во-первых, давайте рассмотрим структуру таблицы, необходимую для построения этой связи:

```
posts  
  id - integer  
  name - string  
  
videos
```

```
id - integer
name - string

tags
id - integer
name - string

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

{Прежде чем погрузиться в полиморфные отношения «многие-ко-многим», вам может быть полезно прочитать документацию по типичным [отношениям «многие-ко-многим»](#).

## Структура модели отношения

### Многие ко многим (полиморфное)

Далее, мы готовы определить отношения в моделях. Обе модели `Post` и `Video` будут содержать метод `tags`, который вызывает метод `morphToMany`, предоставляемый базовым классом модели Eloquent.

Метод `morphToMany` принимает имя связанной модели, а также «имя отношения». В зависимости от имени, которое мы присвоили имени нашей промежуточной таблицы и содержащихся в ней ключей, мы будем называть эту связь `taggable`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Post extends Model
{
    /**
     * Получить все теги поста.
     */
}
```

```
public function tags(): MorphToMany
{
    return $this->morphToMany(Tag::class, 'taggable');
}
```

## Определение обратной связи

### Многие ко многим (полиморфное)

Затем в модели `Tag` вы должны определить метод для каждой из ее возможных родительских моделей. Итак, в этом примере мы определим метод `posts` и метод `videos`. Оба метода должны возвращать результат метода `morphedByMany`.

Метод `morphedByMany` принимает имя связанной модели, а также «имя отношения». В зависимости от имени, которое мы присвоили имени нашей промежуточной таблицы и содержащихся в ней ключей, мы будем называть эту связь `taggable`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Tag extends Model
{
    /**
     * Получить все посты, которым присвоен этот тег.
     */
    public function posts(): MorphToMany
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Получить все видео, которым присвоен этот тег.
     */
    public function videos(): MorphToMany
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

## Получение отношения Многие ко многим (полиморфное)

Как только ваша таблица базы данных и модели определены, вы можете получить доступ к отношениям через свои модели. Например, чтобы получить доступ ко всем тегам для публикации, вы можете использовать динамическое свойство связи `tags`:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->tags as $tag) {
    // ...
}
```

Вы можете получить родительскую модель полиморфного отношения из полиморфной дочерней модели, обратившись к имени метода, который выполняет вызов `morphedByMany`. В данном случае это методы `posts` или `videos` в модели `Tag`:

```
use App\Models\Tag;

$tag = Tag::find(1);

foreach ($tag->posts as $post) {
    // ...
}

foreach ($tag->videos as $video) {
    // ...
}
```

## Именование полиморфных типов

По умолчанию Laravel будет использовать полное имя класса для хранения «типа» связанной модели. Например, учитывая приведенный выше пример отношения «один-ко-многим», где модель `Comment` может принадлежать модели `Post` или `Video`, по умолчанию `commentable_type` будет либо `'App\Models\Post'`, либо `'App\Models\Video'`, соответственно. По желанию можно отделить эти значения от внутренней структуры вашего приложения.

Например, вместо использования названий моделей в качестве «типа» мы можем использовать простые строки, такие как `post` и `video`. Таким образом, значения столбца полиморфного «типа» в нашей базе данных останутся действительными, даже если модели будут переименованы:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);
```

Вы можете зарегистрировать `enforceMorphMap` в методе `boot` вашего класса `App\Providers\AppServiceProvider` или создать отдельный сервис-провайдер для этого, если хотите.

Вы можете определить псевдоним полиморфного типа конкретной модели во время выполнения, используя метод модели `getMorphClass`. И наоборот, вы можете определить полное имя класса, связанное с псевдонимом полиморфного типа, используя метод `Relation::getMorphedModel`:

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

При добавлении «карты полиморфных типов» в существующее приложение каждое значение столбца `*_type` в вашей базе данных, которое все еще содержит полностью определенный класс, необходимо преобразовать в его псевдоним, указанный в «карте полиморфных типов».

## # Динамические отношения

Вы можете использовать метод `resolveRelationUsing` для определения отношений между моделями Eloquent во время выполнения скрипта. Хотя обычно это не рекомендуется для нормальной разработки приложений, но иногда это может быть полезно при разработке пакетов Laravel.

Метод `resolveRelationUsing` принимает желаемое имя отношения в качестве своего первого аргумента. Второй аргумент, передаваемый методу, должен быть замыканием, которое принимает экземпляр модели и возвращает допустимое определение отношения Eloquent. Как правило, вы должны настроить динамические отношения в методе `boot` [поставщика служб](#):

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function (Order $orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

При определении динамических отношений всегда предоставляйте явные аргументы имени ключа методам связи Eloquent.

## # Запросы отношений

Поскольку все отношения Eloquent определяются с помощью методов, вы можете вызывать эти методы для получения экземпляра отношения, не выполняя фактического запроса для загрузки связанных моделей. Кроме того, все типы отношений Eloquent также служат в качестве [построителей запросов](#), позволяя вам продолжать связывать ограничения в запросе отношений, прежде чем окончательно выполнить запрос SQL к вашей базе данных.

Например, представьте себе приложение для блога, в котором модель `User` имеет множество связанных моделей `Post`:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class User extends Model
{
    /**
     * Получить все посты пользователя.
     */
    public function posts(): HasMany
    {
        return $this->hasMany(Post::class);
    }
}
```

Вы можете запросить отношение `posts` и добавить к ним дополнительные ограничения к отношениям, например:

```
use App\Models\User;

$user = User::find(1);

$user->posts()->where('active', 1)->get();
```

Вы можете использовать любой из методов [построителя запросов](#) Laravel для отношения, поэтому обязательно изучите документацию по построителю запросов, чтобы узнать обо всех доступных вам методах.

## Создание цепочки выражений `orWhere` после отношений

Как показано в приведенном выше примере, вы можете добавлять дополнительные ограничения к отношениям при их запросе. Однако, будьте осторожны при создании цепочек выражений `orWhere` с отношением, поскольку предложения `orWhere` будут логически сгруппированы на том же уровне, что и ограничение отношения:

```
$user->posts()
    ->where('active', 1)
    ->orWhere('votes', '>=', 100)
    ->get();
```

В приведенном выше примере будет сгенерирован следующий SQL. Как видите, выражение `or` предписывает запросу возвращать *любой* пост с более чем 100 голосами. Запрос больше не ограничен конкретным пользователем:

```
select *  
from posts  
where user_id = ? and active = 1 or votes >= 100
```

В большинстве ситуаций следует использовать [логические группы](#) для группировки условий в круглые скобки:

```
use Illuminate\Database\Eloquent\Builder;  
  
$user->posts()  
    ->where(function (Builder $query) {  
        return $query->where('active', 1)  
            ->orWhere('votes', '>=', 100);  
    })  
    ->get();
```

В приведенном выше примере будет получен следующий SQL. Обратите внимание, что логическая группировка правильно сгруппировала ограничения, и запрос остается ограниченным для конкретного пользователя:

```
select *  
from posts  
where user_id = ? and (active = 1 or votes >= 100)
```

## Методы отношений против динамических свойств

Если вам не нужно добавлять дополнительные ограничения в запрос отношения Eloquent, вы можете получить доступ к отношению, как если бы это было свойство. Например, продолжая использовать наши модели `User` и `Post` из примера, мы можем получить доступ ко всем постам пользователя следующим образом:

```
use App\Models\User;  
  
$user = User::find(1);  
  
foreach ($user->posts as $post) {
```

```
// ...
}
```

Динамические свойства отношений выполняют «отложенную загрузку», что означает, что они будут загружать данные своих отношений только при фактическом доступе к ним. Из-за этого разработчики часто используют [жадную загрузку](#) для предварительной загрузки отношений, которые, как они знают, будут доступны после загрузки модели. Жадная загрузка обеспечивает значительное сокращение количества SQL-запросов, которые необходимо выполнить для загрузки отношений модели.

## Запрос наличия отношений

При извлечении записей модели бывает необходимо ограничить результаты в зависимости от наличия связи. Например, представьте, что вы хотите получить все посты блога, содержащие хотя бы один комментарий. Для этого вы можете передать имя отношения методам `has` и `orHas`:

```
use App\Models\Post;

// Получить все посты, в которых есть хотя бы один комментарий ...
$post = Post::has('comments')->get();
```

Вы также можете указать оператор и значение счетчика для уточнения запроса:

```
// Получить посты, в которых есть 3 или более комментариев ...
$post = Post::has('comments', '>=', 3)->get();
```

Вы можете использовать «точечную нотацию» для выполнения запроса к вложенным отношениям. Например, вы можете получить все посты, в которых есть хотя бы один комментарий с хотя бы одним изображением:

```
// Получить посты, в которых есть хотя бы один комментарий с изображениями ...
$post = Post::has('comments.images')->get();
```

Если вам нужно еще больше возможностей, вы можете использовать методы `whereHas` и `orWhereHas` для определения дополнительных ограничений запроса для ваших `has`-запросов, например, для проверки содержимого комментария:

```
use Illuminate\Database\Eloquent\Builder;

// Получить посты с хотя бы одним комментарием, содержащим `code%` ...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Получить посты с как минимум десятью комментариями, содержащими `code%` ...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

Eloquent в настоящее время не поддерживает запросы о существовании отношений между базами данных. Отношения должны существовать в одной базе данных.

## Однострочные запросы наличия отношений

Если вы хотите запросить существование отношения с одним простым условием, вам может быть удобнее использовать методы `whereRelation`, `orWhereRelation`, `whereMorphRelation` и `orWhereMorphRelation`. Например, мы можем запросить все посты с не одобренными комментариями:

```
use App\Models\Post;

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

Подобно вызовам метода `where` в построителе запросов, вы также можете явно указать оператор сравнения:

```
$posts = Post::whereRelation(
    'comments', 'created_at', '>=', now()->subHour()
)->get();
```

## Запрос отсутствия отношений

При извлечении записей модели бывает необходимо ограничить результаты на основании отсутствия связи. Например, представьте, что вы хотите получить все посты блога, которые **не** имеют комментариев. Для этого вы можете передать имя отношения методам `doesn'tHave` И `orDoesn'tHave`:

```
use App\Models\Post;

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

Если вам нужно еще больше возможностей, вы можете использовать методы `whereDoesn'tHave` И `orWhereDoesn'tHave` для определения дополнительных ограничений запроса для ваших `doesn'tHave`-запросов, например, для проверки содержимого комментария:

```
use Illuminate\Database\Eloquent\Builder;

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

Вы можете использовать «точечную нотацию» для выполнения запроса к вложенным отношениям. Например, следующий запрос будет извлекать все посты, у которых нет комментариев. Однако, посты с комментариями от авторов, которые не забанены, будут включены в результаты:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesn'tHave('comments.author', function (Builder $query) {
    $query->where('banned', 0);
})->get();
```

## Запрос полиморфных отношений Morph To

Чтобы узнать о существовании полиморфных «один-к» отношений, вы можете использовать методы `whereHasMorph` И `whereDoesn'tHaveMorph`. Эти методы принимают имя отношения в качестве своего первого аргумента. Затем методы принимают имена связанных моделей, которые вы хотите включить в запрос. Наконец, вы можете предоставить замыкание, которое ограничивает запрос отношения:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// Получить комментарии, связанные с постами или видео с заголовком, содержащими `code%
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// Получить комментарии, связанные с постами с заголовком, не содержащим `code%` ...
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class,
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();
```

Иногда требуется добавить ограничения запроса в зависимости от «типа» связанной полиморфной модели. Замыкание, переданное методу `whereHasMorph`, может получить значение `$type` в качестве второго аргумента. Этот аргумент позволяет вам создавать запрос на основе «типа»:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, string $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

Иногда вам может потребоваться запросить дочерние элементы родительского отношения «morph to». Вы можете сделать это, используя методы `whereMorphedTo` и `whereNotMorphedTo`, которые автоматически определят правильное сопоставление

типов морфинга для данной модели. Эти методы принимают имя отношения `morphTo` в качестве первого аргумента и соответствующую родительскую модель в качестве второго аргумента:

```
$comments = Comment::whereMorphedTo('commentable', $post)
    ->orWhereMorphedTo('commentable', $video)
    ->get();
```

## Запрос всех связанных моделей

Допускается использование метасимвола подстановки `*` в качестве значения при передаче массива возможных полиморфных моделей. Это проинструктирует Laravel извлечь все возможные полиморфные типы из базы данных. Laravel выполнит дополнительный запрос, чтобы выполнить эту операцию:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query) {
    $query->where('title', 'like', 'foo%');
})->get();
```

## # Агрегирование связанных моделей

### Подсчет связанных моделей

Иногда требуется подсчитать количество связанных моделей для отношения, не загружая модели. Для этого вы можете использовать метод `withCount`. Метод `withCount` помещает атрибут `{relation}_count` в результирующие модели:

```
use App\Models\Post;

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

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

Передавая массив методу `withCount`, вы можете добавить «счетчики» для нескольких отношений, а также добавить дополнительные ограничения к запросам:

```
use Illuminate\Database\Eloquent\Builder;

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

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

Вы также можете использовать псевдоним результата подсчета отношений, разрешив несколько подсчетов для одной и той же связи:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
])->get();

echo $posts[0]->comments_count;
echo $posts[0]->pending_comments_count;
```

## Отложенная загрузка подсчета связанных моделей

Используя метод `loadCount`, вы можете загрузить счетчик отношений после того, как родительская модель уже была получена:

```
$book = Book::first();

$book->loadCount('genres');
```

Если вам нужно установить дополнительные ограничения запроса для запроса подсчета, вы можете передать массив с ключами отношений, которые вы хотите подсчитать. Значения массива должны быть замыканиями, которые получают экземпляр построителя запросов:

```
$book->loadCount(['reviews' => function (Builder $query) {
    $query->where('rating', 5);
}])
```

## Подсчет отношений и пользовательские операторы SELECT

Если вы комбинируете `withCount` с оператором `SELECT`, убедитесь, что вы вызываете `withCount` после метода `select`:

```
$posts = Post::select(['title', 'body'])
    ->withCount('comments')
    ->get();
```

## Другие агрегатные функции

Помимо метода `withCount`, Eloquent содержит методы `withMin`, `withMax`, `withAvg`, `withSum` и `withExists`. Эти методы добавят атрибут `{relation}_{function}_{column}` в ваши результирующие модели:

```
use App\Models\Post;

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

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

Вы можете явно указать, как должно называться свойство, содержащее результат агрегатной функции, использовав псевдоним:

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

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

Как и метод `loadCount`, также доступны отложенные версии этих методов. Эти дополнительные агрегированные операции могут выполняться на уже полученных моделях Eloquent:

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

Если вы комбинируете эти агрегатные методы с оператором `select`, убедитесь, что вы вызываете их после метода `select`:

```
$posts = Post::select(['title', 'body'])
    ->withExists('comments')
    ->get();
```

## Подсчет связанных моделей отношений Morph To

Если вы хотите загрузить полиморфное отношение «один-к», а также связанные счетчики моделей для различных сущностей, которые могут быть возвращены этим отношением, то вы можете использовать метод `with` в сочетании с отношениями `morphTo` – метод `morphWithCount`.

В этом примере предположим, что модели `Photo` и `Post` могут создавать модели `ActivityFeed`. Предположим, что модель `ActivityFeed` определяет полиморфное отношение «один-к» с именем `parentable`, которое позволяет нам получить родительскую модель `Photo` или `Post` для переданного экземпляра `ActivityFeed`. Кроме того, предположим, что модели `Photo` имеют много моделей `Tag`, а модели `Post` имеют много моделей `Comment`.

Теперь давайте представим, что мы хотим получить экземпляры `ActivityFeed` и загрузить родительские модели для каждого экземпляра `ActivityFeed`. Кроме того, мы хотим получить количество тегов, связанных с каждой родительской фотографией, и количество комментариев, связанных с каждым родительским постом:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
    'parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWithCount([
            Photo::class => ['tags'],
            Post::class => ['comments'],
        ]);
    }
]);
```

```
]);  
}])->get();
```

## Отложенная загрузка подсчета

### связанных моделей отношений Morph To

Предположим, мы уже получили набор моделей `ActivityFeed` и теперь хотим загрузить счетчики вложенных отношений для различных родительских (`parentable`) моделей, связанных с `ActivityFeed`. Для этого вы можете использовать метод `loadMorphCount`:

```
$activities = ActivityFeed::with('parentable')->get();  
  
$activities->loadMorphCount('parentable', [  
    Photo::class => ['tags'],  
    Post::class => ['comments'],  
]);
```

## # Жадная (eager) загрузка

При доступе к отношениям Eloquent как к свойствам, связанные модели загружаются «отложенно». Это означает, что данные отношения фактически не загружаются, пока вы впервые не затребуете доступ к свойству. Однако Eloquent может «жадно» загрузить отношения во время запроса родительской модели. Жадная загрузка позволяет избежать проблем «N+1» с запросами. Чтобы проиллюстрировать проблему «N+1» запроса, рассмотрим модель `Book`, которая «принадлежит» модели `Author`:

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\Relations\BelongsTo;  
  
class Book extends Model  
{  
    /**  
     * Получить автора книги.  
     */
```

```
public function author(): BelongsTo
{
    return $this->belongsTo(Author::class);
}
```

Теперь давайте получим все книги и их авторов:

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Этот цикл выполнит один запрос для получения всех книг из таблицы базы данных, затем еще один запрос для каждой книги, чтобы получить автора книги. Итак, если у нас есть 25 книг, приведенный выше код будет запускать 26 запросов: один для исходной книги и 25 дополнительных запросов для получения автора каждой книги.

К счастью, мы можем использовать жадную загрузку, чтобы сократить эту операцию до двух запросов. При построении запроса вы можете указать, какие отношения должны быть загружены с помощью метода `with`:

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Для этой операции будут выполнены только два запроса – один запрос для получения всех книг и один запрос – для получения всех авторов для всех книг:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

## Жадная загрузка множественных отношений

Иногда вам может понадобиться загрузить несколько разных отношений. Для этого просто передайте массив отношений методу `with`:

```
$books = Book::with(['author', 'publisher'])->get();
```

## Вложенная жадная загрузка

Чтобы жадно загрузить отношения отношений, вы можете использовать «точечную нотацию». Например, давайте загрузим всех авторов книги и все личные контакты авторов:

```
$books = Book::with('author.contacts')->get();
```

В качестве альтернативы вы можете указать вложенные жадно загружаемые отношения, предоставив вложенный массив методу `with`, что может быть удобным при одновременной загрузке нескольких вложенных отношений:

```
$books = Book::with([
    'author' => [
        'contacts',
        'publisher',
    ],
])->get();
```

## Вложенная жадная загрузка отношений Morph To

Если вы хотите загрузить полиморфное отношение «один-к», а также вложенные отношения для различных сущностей, которые могут быть возвращены этим отношением, то вы можете использовать метод `with` в сочетании с отношениями `morphTo` – метод `morphWith`. Чтобы проиллюстрировать этот метод, давайте рассмотрим следующую модель:

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
```

```
/**  
 * Получить родительский элемент записи ленты активности.  
 */  
public function parentable(): MorphTo  
{  
    return $this->morphTo();  
}  
}
```

В этом примере предположим, что модели `Event`, `Photo` и `Post` могут создавать модели `ActivityFeed`. Кроме того, предположим, что модели `Event` принадлежат модели `Calendar`, модели `Photo` связаны с моделями `Tag`, а модели `Post` принадлежат модели `Author`.

Используя эти определения моделей и отношения, мы можем получить экземпляры модели `ActivityFeed` и жадно загрузить все родительские (`parentable`) модели и их соответствующие вложенные отношения:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;  
  
$activities = ActivityFeed::query()  
->with(['parentable' => function (MorphTo $morphTo) {  
    $morphTo->morphWith([  
        Event::class => ['calendar'],  
        Photo::class => ['tags'],  
        Post::class => ['author'],  
    ]);  
}])->get();
```

## Жадная загрузка определенных столбцов

Вам не всегда может понадобиться каждый столбец из извлекаемых вами отношений. По этой причине Eloquent позволяет вам указать, какие столбцы отношения вы хотите получить:

```
$books = Book::with('author:id,name,book_id')->get();
```

При использовании этого функционала вы всегда должны включать столбец `id` и любые

соответствующие столбцы внешнего ключа в список столбцов, которые вы хотите получить.

## Жадная загрузка по умолчанию

Иногда требуется постоянная загрузка некоторых отношений при извлечении модели. Для этого вы можете определить свойство `$with` в модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
    /**
     * Отношения, которые всегда должны быть загружены.
     *
     * @var array
     */
    protected $with = ['author'];

    /**
     * Получить автора книги.
     */
    public function author(): BelongsTo
    {
        return $this->belongsTo(Author::class);
    }

    /**
     * Получить жанр книги
     */
    public function genre(): BelongsTo
    {
        return $this->belongsTo(Genre::class);
    }
}
```

Если вы хотите удалить элемент из свойства `$with` для одного запроса, вы можете использовать метод `without`:

```
$books = Book::without('author')->get();
```

Если вы хотите переопределить все элементы свойства `$with` для одного запроса, вы можете использовать метод `withOnly`:

```
$books = Book::withOnly('genre')->get();
```

## Ограничение жадной загрузки

Иногда требуется жадная загрузка отношения с указанием дополнительного условия. Вы можете сделать это, передав массив отношений методу `with`, где ключ массива – это имя отношения, а значение массива – это функция, которая добавляет дополнительные ограничения к запросу жадной загрузки:

```
use App\Models\User;
use Illuminate\Contracts\Database\Eloquent\Builder;

$users = User::with(['posts' => function (Builder $query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

В этом примере Eloquent будет загружать только те посты, столбец `title` которых содержит слово `code`. Вы можете использовать и другие методы [построителя запросов](#):

```
$users = User::with(['posts' => function (Builder $query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

## Ограничение жадной загрузки отношений Morph To

Если вы хотите жадно загрузить полиморфное отношение «один-к», Eloquent выполнит несколько запросов для получения каждого типа связанной модели. Вы можете добавить дополнительные ограничения к каждому из этих запросов, используя метод `constrain` полиморфного отношения «один-к»:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
    $morphTo->constrain([
        Post::class => function ($query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function ($query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

В этом примере Eloquent будет загружать только те посты, которые не были скрыты, а видео только с типом как образовательное.

## Ограничение жадной загрузки наличием отношений

Иногда вам может потребоваться проверить наличие отношения, одновременно загружая отношение на основе тех же условий. Например, вы можете захотеть получить только модели `User`, у которых есть дочерние модели `Post`, соответствующие определенному условию запроса, загружая при этом соответствующие посты. Это можно сделать с использованием метода `withWhereHas`:

```
use App\Models\User;

$users = User::withWhereHas('posts', function ($query) {
    $query->where('featured', true);
})->get();
```

## Жадная ПОСТ-загрузка

Иногда требуется жадно загрузить отношение только после получения родительской модели. Например, это может быть полезно, если вам нужно динамически решать, загружать ли связанные модели:

```
use App\Models\Book;

$books = Book::all();
```

```
if ($someCondition) {  
    $books->load('author', 'publisher');  
}
```

Если вам нужно задать дополнительные ограничения запроса жадной загрузки, вы можете передать массив с ключом отношений, которые вы хотите загрузить. Значения массива должны быть экземплярами замыкания, которые получают экземпляр запроса:

```
$author->load(['books' => function (Builder $query) {  
    $query->orderBy('published_date', 'asc');  
}]);
```

Чтобы загрузить отношение только в том случае, если оно еще не было загружено, используйте метод `loadMissing`:

```
$book->loadMissing('author');
```

## Вложенная жадная пост-загрузка и отношения Morph To

Если вы хотите жадно загрузить полиморфное отношение «один-к», а также вложенные отношения для различных сущностей, которые могут быть возвращены этим отношением, вы можете использовать метод `loadMorph`.

Этот метод принимает имя полиморфного отношения «один-к» в качестве своего первого аргумента и массив пар модель / отношение в качестве второго аргумента. Чтобы проиллюстрировать этот метод, давайте рассмотрим следующую модель:

```
<?php
```

```
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\Relations\MorphTo;  
  
class ActivityFeed extends Model  
{  
    /**  
     * Получить родительский элемент записи ленты активности.  
     */  
    public function parentable(): MorphTo  
    {
```

```
        return $this->morphTo();
    }
}
```

В этом примере предположим, что модели `Event`, `Photo` И `Post` могут создавать модели `ActivityFeed`. Кроме того, предположим, что модели `Event` принадлежат модели `Calendar`, модели `Photo` связаны с моделями `Tag`, а модели `Post` принадлежат модели `Author`.

Используя эти определения моделей и отношения, мы можем получить экземпляры модели `ActivityFeed` и жадно загрузить все родительские (`parentable`) модели и их соответствующие вложенные отношения:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
});
```

## Предотвращение ленивой загрузки

Как уже говорилось ранее, жадная загрузка отношений часто может обеспечить значительный выигрыш в производительности. Поэтому, если вы хотите, вы можете указать Laravel всегда предотвращать ленивую загрузку отношений. Для этого вы можете вызвать метод `preventLazyLoading`, предлагаемый базовым классом модели Eloquent. Обычно этот метод вызывается в методе `boot` класса `AppServiceProvider` вашего приложения.

Метод `preventLazyLoading` принимает необязательный аргумент `boolean`, который указывает, следует ли предотвратить ленивую загрузку. Например, вы можете отключить ленивую загрузку только в девелопмент-среде, чтобы ваша продакшн-среда продолжала нормально функционировать, если лениво загруженные отношения случайно присутствуют в коде на сервере:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Bootstrap any application services.
 */

```

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

После отключения ленивой загрузки Eloquent будет выбрасывать исключение `Illuminate\Database\LazyLoadingViolationException`, когда ваше приложение попытается лениво загрузить любое отношение Eloquent.

Вы можете настроить это поведение с помощью метода `handleLazyLoadingViolationsUsing`. Например, используя этот метод, вы можете указать, что нарушения надо только регистрировать, а не выбрасывать исключение:

```
Model::handleLazyLoadingViolationUsing(function (Model $model, string $relation) {
    $class = $model::class;

    info("Attempted to lazy load [{$relation}] on model [{$class}].");
});
```

## # Вставка и обновление связанных моделей

### Метод Save

Eloquent содержит удобные методы для добавления новых моделей в отношения. Например, возможно, вам нужно добавить новый комментарий к посту. Вместо того чтобы вручную задавать атрибут `post_id` в модели `Comment`, вы можете вставить комментарий, используя метод отношения `save`:

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

Обратите внимание, что мы не обращались к связи `comments` как к динамическому свойству. Вместо этого мы вызвали метод `comments`, чтобы получить экземпляр отношения. Метод `save` автоматически добавит соответствующее значение `post_id` в новую модель `Comment`.

Если вам нужно сохранить несколько связанных моделей, вы можете использовать метод `saveMany`:

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
```

Методы `save` и `saveMany` не будут добавлять новые модели ни в какие отношения, хранимые в памяти, прежде загруженные в родительскую модель. Если вы планируете получить доступ к отношениям после использования методов `save` или `saveMany`, то вы можете использовать метод `refresh` для перезагрузки модели и ее отношений:

```
$post->comments()->save($comment);

$post->refresh();

// Все комментарии, включая только что сохраненный комментарий ...
$post->comments;
```

## Рекурсивное сохранение моделей и отношений

Если вы хотите сохранить вашу модель и все связанные с ней отношения, вы можете использовать метод `push`. В этом примере будет сохранена модель `Post`, а также ее комментарии и авторы этих комментариев:

```
$post = Post::find(1);

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

Метод `pushQuietly` может быть использован для сохранения модели и ее связанных отношений без вызова каких-либо событий:

```
$post->pushQuietly();
```

## Метод Create

В дополнение к методам `save` и `saveMany` вы также можете использовать метод `create`, который принимает массив атрибутов, создает модель и вставляет ее в базу данных. Разница между `save` и `create` в том, что `save` принимает полный экземпляр модели Eloquent, а `create` принимает простой массив PHP. Вновь созданная модель будет возвращена методом `create`:

```
use App\Models\Post;

$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Вы можете использовать метод `createMany` для создания нескольких связанных моделей:

```
$post = Post::find(1);

$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

Методы `createQuietly` и `createManyQuietly` могут быть использованы для создания модели(ей) без отправки каких-либо событий:

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

$user->posts()->createQuietly([
    'title' => 'Post title.',
]);
```

```
$user->posts()->createManyQuietly([
    ['title' => 'First post.'],
    ['title' => 'Second post.'],
]);
```

Вы также можете использовать методы `findOrNew`, `firstOrNew`, `firstOrCreate`, и `updateOrCreate` для [создания и обновления моделей отношений](#).

Перед использованием метода `create` обязательно ознакомьтесь с документацией о [массовом присвоении](#) атрибутов.

## Обновление отношений Один К

Если вы хотите назначить дочернюю модель новой родительской модели, вы можете использовать метод `associate`. В этом примере модель `User` определяет отношение `belongsTo` к модели `Account`. Метод `associate` установит внешний ключ дочерней модели:

```
use App\Models\Account;

$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

Чтобы удалить родительскую модель из дочерней модели, вы можете использовать метод `dissociate`. Этот метод установит для внешнего ключа отношения значение `null`:

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

$user->save();
```

## Обновление отношений Многие ко многим

## Присоединение / Отсоединение

### отношений Многие ко многим

Eloquent также содержит методы, которые делают работу с отношениями «многие-ко-многим» более удобной. Например, представим, что у пользователя может быть много ролей, а у роли может быть много пользователей. Вы можете использовать метод `attach`, чтобы присоединить роль к пользователю, вставив запись в промежуточную таблицу отношения:

```
use App\Models\User;

$user = User::find(1);

$user->roles()->attach($roleId);
```

При присоединении отношения к модели вы также можете передать массив дополнительных данных для вставки в промежуточную таблицу:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Иногда может потребоваться удалить роль пользователя. Чтобы удалить запись отношения «многие-ко-многим», используйте метод `detach`. Метод `detach` удалит соответствующую запись из промежуточной таблицы; однако обе модели останутся в базе данных:

```
// Отсоединяем одну роль от пользователя ...
$user->roles()->detach($roleId);

// Отсоединяем от пользователя все роли ...
$user->roles()->detach();
```

Для удобства `attach` и `detach` также принимают в качестве входных данных массивы идентификаторов:

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

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
```

```
1 => ['expires' => $expires],  
2 => ['expires' => $expires],  
]);
```

## Синхронизация ассоциаций отношений Многие ко многим

Вы также можете использовать метод `sync` для построения ассоциаций «многие-ко-многим». Метод `sync` принимает массив идентификаторов для размещения в промежуточной таблице. Любые идентификаторы, которых нет в данном массиве, будут удалены из промежуточной таблицы. После завершения этой операции в промежуточной таблице будут существовать только ID из переданного массива:

```
$user->roles()->sync([1, 2, 3]);
```

Вы также можете вместе с ID передать дополнительные значения промежуточной таблицы:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

Если вы хотите вставить одинаковые значения в промежуточную таблицу для каждого ID, вы можете использовать метод `syncWithPivotValues`:

```
$user->roles()->syncWithPivotValues([1, 2, 3], ['active' => true]);
```

Если вы не хотите удалять существующие связи, идентификаторы которых отсутствуют в переданном массиве, то вы можете использовать метод `syncWithoutDetaching`:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

## Переключение ассоциаций отношений Многие ко многим

Отношение «многие-ко-многим» также содержит метод `toggle`, который «переключает» статус присоединения указанных идентификаторов связанных моделей. Если переданный идентификатор в настоящее время присоединен, он будет отсоединен. Аналогично, если он в настоящее время отсоединен, то он будет присоединен:

```
$user->roles()->toggle([1, 2, 3]);
```

Вы также можете передать дополнительные значения для промежуточной таблицы вместе с идентификаторами:

```
$user->roles()->toggle([
    1 => ['expires' => true],
    2 => ['expires' => true],
]);
```

## Обновление записи сводной таблицы отношений Многие ко многим

Если вам нужно обновить существующую строку в промежуточной таблице ваших отношений, то вы можете использовать метод `updateExistingPivot`. Этот метод принимает внешний ключ промежуточной записи и массив атрибутов для обновления:

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

$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

## # Затрагивание временных меток родителя

Когда в модели определены методы `belongsTo` или `belongsToMany` по отношению к другой модели, например `Comment`, который принадлежит `Post`, то иногда бывает необходимо обновить временную метку родителя при обновлении дочерней модели.

Например, когда модель `Comment` обновляется, то вы можете автоматически «затронуть» временную метку `updated_at` родительской модели `Post`, чтобы она была установлена на текущую дату и время. Для этого вы можете добавить свойство `$touches` к дочерней модели, содержащее имена отношений, для которых должны обновляться временные метки `updated_at` при обновлении дочерней модели:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * Все отношения, временные метки которых должны быть затронуты.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Получить пост, к которому принадлежит комментарий.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Временные метки родительской модели будут обновлены только в том случае, если дочерняя модель обновлена с помощью метода `save` Eloquent.

# Eloquent · Коллекции

# Введение

# Доступные методы

# Пользовательские коллекции

## # Введение

Все методы Eloquent, которые возвращают более одного результата модели, будут возвращать экземпляры класса `Illuminate\Database\Eloquent\Collection`, включая результаты, полученные с помощью метода `get` или доступные через отношения. Объект коллекции Eloquent расширяет [базовую коллекцию](#) Laravel, поэтому он естественным образом наследует десятки методов, использующих в работе текущий интерфейс с базовым массивом моделей Eloquent. Обязательно ознакомьтесь с документацией по коллекции Laravel, чтобы узнать все об этих полезных методах!

Все коллекции также являются итераторами, что позволяет вам перебирать их, как если бы они были простыми массивами PHP:

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

Однако, как упоминалось ранее, коллекции намного мощнее массивов и предоставляют множество методов типа `map` / `reduce`, которые могут быть связаны с помощью интуитивно понятного интерфейса. Например, мы можем удалить все неактивные модели, а затем собрать имена оставшихся пользователей:

```
$names = User::all()->reject(function (User $user) {
    return $user->active === false;
```

```
})->map(function (User $user) {
    return $user->name;
});
```

## Преобразование коллекций Eloquent

В то время как большинство методов коллекции Eloquent возвращают новый экземпляр коллекции Eloquent, методы `collapse`, `flatten`, `flip`, `keys`, `pluck`, и `zip` возвращают экземпляр [базовой коллекции](#). Аналогично, если метод `map` возвращает коллекцию, не содержащую никаких моделей Eloquent, она будет преобразована в экземпляр базовой коллекции.

## # Доступные методы

Все коллекции Eloquent расширяют базовый класс [коллекций Laravel](#); поэтому они наследуют все мощные методы, предоставляемые классом базовой коллекции.

Кроме того, класс `Illuminate\Database\Eloquent\Collection` содержит расширенный набор методов, помогающих управлять коллекциями моделей. Большинство методов возвращают экземпляры `Illuminate\Database\Eloquent\Collection`; однако некоторые методы, такие как `modelKeys`, возвращают экземпляр `Illuminate\Support\Collection`.

[append](#)

[contains](#)

[diff](#)

[except](#)

[find](#)

[fresh](#)

[intersect](#)

[load](#)

[loadMissing](#)

[modelKeys](#)

[makeVisible](#)

[makeHidden](#)

[only](#)

[setVisible](#)

[setHidden](#)

[toQuery](#)

[unique](#)

## append(\$attributes)

Метод [append](#) может быть использован для указания, что атрибут должен быть [добавлен](#) к каждой модели в коллекции. Этот метод принимает массив атрибутов или один атрибут:

```
$users->append('team');
```

```
$users->append(['team', 'is_admin']);
```

## contains(\$key, \$operator = null, \$value = null)

Метод [contains](#) используется для определения того, содержится ли переданный экземпляр модели в коллекции. Этот метод принимает первичный ключ или экземпляр модели:

```
$users->contains(1);
```

```
$users->contains(User::find(1));
```

## diff(\$items)

Метод [diff](#) возвращает все модели, которых нет в переданной коллекции:

```
use App\Models\User;
```

```
$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());
```

## except(\$keys)

Метод [except](#) возвращает все модели, у которых нет переданных первичных ключей:

```
$users = $users->except([1, 2, 3]);
```

## find(\$key)

Метод `find` возвращает модель, у которой есть первичный ключ, соответствующий переданному ключу. Если `$key` является экземпляром модели, `find` попытается вернуть модель, соответствующую первичному ключу. Если `$key` является массивом ключей, `find` вернет все модели, у которых есть первичный ключ в переданном массиве:

```
$users = User::all();  
  
$user = $users->find(1);
```

## findOrFail(\$key)

Метод `findOrFail` возвращает модель, первичный ключ которой соответствует заданному ключу, или выдает исключение `Illuminate\Database\Eloquent\ModelNotFoundException`, если в коллекции не найдена соответствующая модель:

```
$users = User::all();  
  
$user = $users->findOrFail(1);
```

## fresh(\$with = [])

Метод `fresh` извлекает из базы данных свежий экземпляр каждой модели в коллекции. Кроме того, будут загружены любые указанные отношения:

```
$users = $users->fresh();  
  
$users = $users->fresh('comments');
```

## intersect(\$items)

Метод `intersect` возвращает все модели, которые также присутствуют в переданной коллекции:

```
use App\Models\User;

$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());
```

## load(\$relations)

Метод `load` нетерпеливо загружает указанные отношения для всех моделей в коллекции:

```
$users->load(['comments', 'posts']);

$users->load('comments.author');

$users->load(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

## loadMissing(\$relations)

Метод `loadMissing` нетерпеливо загружает указанные отношения для всех моделей в коллекции, если отношения еще не загружены:

```
$users->loadMissing(['comments', 'posts']);

$users->loadMissing('comments.author');

$users->loadMissing(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

## modelKeys()

Метод `modelKeys` возвращает первичные ключи для всех моделей в коллекции:

```
$users->modelKeys();

// [1, 2, 3, 4, 5]
```

## makeVisible(\$attributes)

Метод `makeVisible` делает видимыми атрибуты, которые обычно «скрыты» для каждой модели коллекции:

```
$users = $users->makeVisible(['address', 'phone_number']);
```

## makeHidden(\$attributes)

Метод `makeHidden` скрывает атрибуты, которые обычно «видны» для каждой модели в коллекции:

```
$users = $users->makeHidden(['address', 'phone_number']);
```

## only(\$keys)

Метод `only` возвращает все модели с указанными первичными ключами:

```
$users = $users->only([1, 2, 3]);
```

## setVisible(\$attributes)

Метод `setVisible` временно переопределяет видимые атрибуты для каждой модели в коллекции:

```
$users = $users->setVisible(['id', 'name']);
```

## setHidden(\$attributes)

Метод `setHidden` временно переопределяет скрытые атрибуты для каждой модели в коллекции:

```
$users = $users->setHidden(['email', 'password', 'remember_token']);
```

## toQuery()

Метод `toQuery` возвращает экземпляр построителя запросов Eloquent, содержащий ограничение `whereIn` для первичных ключей модели коллекции:

```
use App\Models\User;
```

```
$users = User::where('status', 'VIP')->get();  
  
$users->toQuery()->update([  
    'status' => 'Administrator',  
]);
```

## unique(\$key = null, \$strict = false)

Метод `unique` возвращает все уникальные модели в коллекции. Любые модели с тем же первичным ключом, что и другая модель в коллекции, удаляются:

```
$users = $users->unique();
```

## # Пользовательские коллекции

Если вы хотите использовать собственный объект `Collection` при взаимодействии с конкретной моделью, вы можете добавить атрибут `CollectedBy` к своей модели:

```
<?php  
  
namespace App\Models;  
  
use App\Support\UserCollection;  
use Illuminate\Database\Eloquent\Attributes\CollectedBy;  
use Illuminate\Database\Eloquent\Model;  
  
#[CollectedBy(UserCollection::class)]  
class User extends Model  
{  
    // ...  
}
```

Альтернативно вы можете определить метод `newCollection` в своей модели:

```
<?php  
  
namespace App\Models;  
  
use App\Support\UserCollection;  
use Illuminate\Database\Eloquent\Collection;  
use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
{
    /**
     * Создать новый экземпляр коллекции Eloquent.
     *
     * @param array<int, \Illuminate\Database\Eloquent\Model> $models
     * @return \Illuminate\Database\Eloquent\Collection<int, \Illuminate\Database\Elo-
     */
    public function newCollection(array $models = []): Collection
    {
        return new UserCollection($models);
    }
}
```

После того как вы определили метод `newCollection` или добавили атрибут `CollectedBy` в свою модель, вы будете получать экземпляр своей пользовательской коллекции в любое время, когда Eloquent обычно возвращает экземпляр `\Illuminate\Database\Eloquent\Collection`.

Если вы хотите использовать собственную коллекцию для каждой модели в вашем приложении, вы должны определить метод `newCollection` для класса базовой модели, который расширяется всеми моделями вашего приложения.

# Eloquent · Мутаторы и типизация

## # Введение

## # Аксессоры и мутаторы (Accessors and Mutators)

# Определение аксессора (Accessor)

# Определение мутатора (Mutator)

## # Приведение атрибутов к типам

# Преобразование в массив и JSON

# Типизация даты

# Типизация "Enum"

# Типизация "Encrypted"

# Типизация во время запроса

## # Пользовательская типизация

# Типизация объект-значение

# Сериализация в массив и JSON

# Входящая типизация

# Параметры типизации

# Интерфейс Castable

## # Введение

Аксессоры, мутаторы и приведение атрибутов к типам позволяют преобразовывать значения атрибутов Eloquent, когда вы извлекаете экземпляр модели или присваиваете их экземпляру модели. Например, вы можете использовать [шифровальщик Laravel](#), чтобы зашифровать значение при его сохранении в базу данных, а затем автоматически расшифровать атрибут при доступе к нему в модели Eloquent. Или вы можете преобразовать строку JSON, которая хранится в вашей базе данных, в массив при доступе к ней через вашу модель Eloquent.

## # Аксессоры и мутаторы (Accessors and Mutators)

# Определение аксессора (Accessor)

Аксессор преобразует значение атрибута экземпляра Eloquent при обращении к нему. Чтобы определить аксессор, создайте `protected` метод в вашей модели, представляющий соответствующий атрибут. Имя этого метода должно соответствовать “camel case” представлению фактического атрибута модели / столбца базы данных, когда это применимо.

В этом примере мы определим аксессор для атрибута `first_name`. Этот аксессор будет автоматически вызываться Eloquent при попытке получения значения атрибута `first_name`. Все методы аксессоров и мутаторов атрибутов должны объявлять тип возвращаемого значения `Illuminate\Database\Eloquent\Cast\Attribute`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Получить имя пользователя.
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
        );
    }
}
```

Все аксессоры возвращают экземпляр `Attribute`, который определяет, как будет осуществлен доступ к атрибуту и, при необходимости, его мутация. В данном примере мы определяем только способ доступа к атрибуту. Для этого мы передаем аргумент `get` конструктору класса `Attribute`.

Как видите, исходное значение столбца передается аксессору, что позволяет вам манипулировать и возвращать значение. Чтобы получить доступ к значению аксессора, вы можете просто получить доступ к атрибуту `first_name` экземпляра модели:

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```

Если вы хотите, чтобы эти вычисленные значения были добавлены к представлениям массива / JSON вашей модели, вам нужно будет добавить их.

## Создание объектов-значений из нескольких атрибутов

Иногда вашему аксессору может потребоваться преобразовать несколько атрибутов модели в один “объект-значение” (value object). Для этого ваше замыкание `get` может принимать второй аргумент `$attributes`, который будет автоматически передаваться в замыкание и будет содержать массив всех текущих атрибутов модели:

```
use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * Взаимодействуйте с адресом пользователя.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
    );
}
```

## Кэширование аксессоров

При возвращении объектов-значений из аксессоров любые изменения, внесенные в объект-значение, автоматически синхронизируются с моделью перед ее сохранением. Это возможно благодаря тому, что Eloquent сохраняет экземпляры,

возвращаемые аксессорами, чтобы каждый раз при вызове аксессора возвращать тот же экземпляр:

```
use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Line 1 Value';
$user->address->lineTwo = 'Updated Address Line 2 Value';

$user->save();
```

Однако иногда вам может потребоваться включить кэширование для примитивных значений, таких как строки и логические значения, особенно если они требуют больших вычислительных ресурсов. Для этого вы можете вызвать метод `shouldCache` при определении вашего аксессора:

```
protected function hash(): Attribute
{
    return Attribute::make(
        get: fn (string $value) => bcrypt(gzuncompress($value)),
        )->shouldCache();
}
```

Если вы хотите отключить кэширование объектов для атрибутов, вы можете вызвать метод `withoutObjectCaching` при определении атрибута:

```
/**
 * Взаимодействуйте с адресом пользователя.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
        )->withoutObjectCaching();
}
```

## Определение мутатора (Mutator)

Мутатор преобразует значение атрибута в момент их присвоения экземпляру Eloquent. Чтобы определить мутатор, вы можете использовать аргумент `set` при определении вашего атрибута.

Определим мутатор для атрибута `first_name`. Этот мутатор будет автоматически вызываться, когда мы попытаемся присвоить значение атрибута `first_name` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Манипуляции с именем пользователя
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
            set: fn (string $value) => strtolower($value),
        );
    }
}
```

Замыкание мутатора получит значение, заданное для атрибута, что позволит вам манипулировать этим значением и возвращать измененное значение. Чтобы использовать наш мутатор, нам нужно только установить атрибут `first_name` для модели Eloquent:

```
use App\Models\User;

$user = User::find(1);

$user->first_name = 'Sally';
```

В этом примере замыкание `set` будет вызываться со значением `Sally`. Затем, мутатор применит к имени функцию `strtolower` и установит полученное значение во внутреннем массиве `$attributes`.

## Мутация нескольких атрибутов

Иногда вашему мутатору может потребоваться установить несколько атрибутов в основной модели. Для этого вы можете вернуть массив из замыкания `set`. Каждый ключ в массиве должен соответствовать базовому атрибуту / столбцу базы данных, связанному с моделью:

```
use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * Манипуляции с адресом пользователя.
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
        set: fn (Address $value) => [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ],
    );
}
```

## # Приведение атрибутов к типам

Приведение атрибутов обеспечивает функциональность, аналогичную аксессорам и мутаторам, но без необходимости определения каких-либо дополнительных методов вашей модели. Вместо этого метод `casts` вашей модели представляет удобный способ преобразования атрибутов в общие типы данных.

Метод `casts` должен возвращать массив, где ключом является имя приводимого атрибута, а значением — тип, к которому вы хотите привести столбец.

Поддерживаемые типы преобразования:

- `array`
- `AsStringable::class`

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

Чтобы продемонстрировать преобразование атрибутов, давайте преобразуем атрибут `is_admin`, который хранится в нашей базе данных в виде целого числа (`0` или `1`), в логическое значение:

```
<?php
```

```
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
{
    /**
     * Получение атрибутов, которые должны быть типизированы.
     *
     * @return array<string, string>
     */
    protected function casts(): array
    {
        return [
            'is_admin' => 'boolean',
        ];
    }
}
```

После определения типизации, атрибут `is_admin` всегда будет преобразован в логическое значение при доступе к нему, даже если базовое значение хранится в базе данных как целое число:

```
$user = App\Models\User::find(1);

if ($user->is_admin) {
    // ...
}
```

Если вам нужно добавить новое временное приведение во время выполнения, вы можете использовать метод `mergeCasts`. Эти определения приведения будут добавлены к любому из уже определенных для модели приведения:

```
$user->mergeCasts([
    'is_admin' => 'integer',
    'options' => 'object',
]);
```

Атрибуты, которые имеют значение `null`, не будут преобразованы. Кроме того, вы никогда не должны определять типизацию (или атрибут), имя которого совпадает с именем отношения.

## Преобразование в строку

Вы можете использовать класс приведения `Illuminate\Database\Eloquent\Cast\AsStringable` для приведения атрибута модели к объекту [строки Fluent Illuminate\Support\Stringable](#):

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\AsStringable;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Получение атрибутов, которые должны быть типизированы.
     *
     * @return array<string, string>
     */
    protected function casts(): array
    {
        return [
            'directory' => AsStringable::class,
        ];
    }
}
```

## Преобразование в массив и JSON

Преобразование в `array` особенно полезно при работе со столбцами, которые хранятся как сериализованный JSON. Например, если ваша база данных имеет поле типа `JSON` или `TEXT`, содержащее сериализованный JSON, то добавленная типизация `array` этому атрибуту автоматически десериализует атрибут модели Eloquent в массив PHP при обращении к нему:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
```

```
/**  
 * Получение атрибутов, которые должны быть типизированы.  
 *  
 * @return array<string, string>  
 */  
protected function casts(): array  
{  
    return [  
        'options' => 'array',  
    ];  
}  
}
```

Как только типизация определена, вы можете получить доступ к атрибуту `options`, и он будет автоматически десериализован из JSON в массив PHP. Когда вы устанавливаете значение атрибута `options`, данный массив будет автоматически сериализован обратно в JSON для сохранения:

```
use App\Models\User;  
  
$user = User::find(1);  
  
$options = $user->options;  
  
$options['key'] = 'value';  
  
$user->options = $options;  
  
$user->save();
```

Чтобы обновить одно поле JSON-атрибута с помощью краткого синтаксиса, вы можете [разрешить массовое назначение](#) и использовать оператор `->` при вызове метода `update`:

```
$user = User::find(1);  
  
$user->update(['options->key' => 'value']);
```

## Типизация ArrayObject и Collection

Хотя типизации стандартного `array` достаточно для многих приложений, но у него есть некоторые недостатки. Поскольку типизация `array` возвращает примитивный

типа, невозможно напрямую изменить смещение массива. Например, следующий код вызовет ошибку PHP:

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

$user->options['key'] = $value;
```

Чтобы решить эту проблему, Laravel предлагает типизацию [AsArrayObject](#), которая преобразует ваш атрибут JSON в класс [ArrayObject](#). Эта функция реализована с использованием реализации [пользовательской типизации](#) Laravel, которая позволяет Laravel интеллектуально кешировать и преобразовывать измененный объект таким образом, что отдельные смещения могли быть изменены без ошибок PHP. Чтобы использовать типизацию [AsArrayObject](#), просто назначьте его атрибуту:

```
use Illuminate\Database\Eloquent\Cast\AsArrayObject;

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [
        'options' => AsArrayObject::class,
    ];
}
```

Точно так же Laravel предлагает типизацию [AsCollection](#), которая преобразует ваш атрибут JSON в экземпляр Laravel [Collection](#):

```
use Illuminate\Database\Eloquent\Cast\AsCollection;

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [

```

```
        'options' => AsCollection::class,
    ];
}
```

Если вы хотите, чтобы приведение типа `AsCollection` создавало экземпляр пользовательского класса коллекции вместо базового класса коллекции Laravel, вы можете указать имя класса коллекции в качестве аргумента приведения типа:

```
use App\Collections\OptionCollection;
use Illuminate\Database\Eloquent\Casts\AsCollection;

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [
        'options' => AsCollection::using(OptionCollection::class),
    ];
}
```

## Типизация даты

По умолчанию Eloquent преобразует столбцы `created_at` и `updated_at` в экземпляры [Carbon](#), расширяющего класс `DateTime` PHP и предоставляющего набор полезных методов. Вы можете типизировать дополнительные атрибуты даты, определив дополнительные преобразования даты в методе вашей модели `cast`. Обычно даты следует приводить с использованием типизации `datetime` или `immutable_datetime`.

При определении типизации `date` или `datetime` вы также можете указать формат даты. Этот формат будет использоваться, когда [модель сериализуется в массив или JSON](#):

```
/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
```

```
        return [
            'created_at' => 'datetime:Y-m-d',
        ];
    }
```

Когда столбец типизирован как дата, вы можете установить соответствующее значение атрибута модели в виде временной метки форматов UNIX, строки даты (`Y-m-d`), строки даты-времени или экземпляров `DateTime` / `Carbon`. Значение даты будет правильно преобразовано и сохранено в вашей базе данных.

Вы можете настроить формат сериализации по умолчанию для всех дат вашей модели, переопределив метод `serializeDate` вашей модели. Этот метод не влияет на форматирование дат для их сохранения в базе данных:

```
/**
 * Подготовить дату для сериализации массива / JSON.
 */
protected function serializeDate(DateTimeInterface $date): string
{
    return $date->format('Y-m-d');
}
```

Чтобы указать формат, который следует использовать при фактическом сохранении дат модели в вашей базе данных, вы должны определить свойство `$dateFormat` вашей модели:

```
/**
 * Формат хранения столбцов даты модели.
 *
 * @var string
 */
protected $dateFormat = 'U';
```

## Приведение даты, сериализация и часовые пояса

По умолчанию приведение `date` и `datetime` будут сериализовывать даты в строку даты UTC ISO-8601 (`YYYY-MM-DDTHH:MM:SS.uuuuuuZ`), независимо от часового пояса, указанного в конфигурации `timezone` вашего приложения. Вам настоятельно рекомендуется всегда использовать этот формат сериализации, а также хранить даты вашего приложения в часовом поясе UTC, не изменяя параметр

конфигурации вашего приложения `timezone` от значения по умолчанию `UTC`.

Последовательное использование часового пояса UTC во всем приложении обеспечит максимальный уровень взаимодействия с другими библиотеками обработки даты, написанными на PHP и JavaScript.

Если к приведению `date` или `datetime` применяется настраиваемый формат, такой как `datetime:Y-m-d H:i:s`, внутренний часовой пояс экземпляра Carbon будет использоваться во время сериализации даты. Обычно это часовой пояс, указанный в параметре конфигурации вашего приложения `timezone`. Однако важно отметить, что столбцы `timestamp`, такие как `created_at` и `updated_at`, не подвержены этому поведению и всегда форматируются в формате UTC, независимо от настроек часового пояса приложения.

## Типизация "Enum"

Eloquent также позволяет вам преобразовывать значения ваших атрибутов в перечисления PHP. Для этого вы можете указать атрибут и перечисление, которое вы хотите преобразовать, в методе вашей модели`casts`:

```
use App\Enums\ServerStatus;

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [
        'status' => ServerStatus::class,
    ];
}
```

После того как вы определили приведение в своей модели, указанный атрибут будет автоматически преобразован в перечисление и из него, когда вы взаимодействуете с атрибутом:

```
if ($server->status == ServerStatus::Provisioned) {
    $server->status = ServerStatus::Provisioned;
```

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

## Типизация массивов перечислений

Иногда вам может потребоваться, чтобы ваша модель сохраняла массив значений перечисления в одном столбце. Для этого вы можете воспользоваться приведением `AsEnumArrayObject` или `AsEnumCollection`, предоставленными Laravel:

```
use App\Enums\ServerStatus; use  
Illuminate\Database\Eloquent\Casts\AsEnumCollection;
```

```
/**  
 * Получение атрибутов, которые должны быть типизированы.  
 *  
 * @return array<string, string>  
 */  
protected function casts(): array  
{  
    return [  
        'statuses' => AsEnumCollection::of(ServerStatus::class),  
    ];  
}
```

## Типизация "Encrypted"

Приведение `encrypted` зашифрует значение атрибута модели, используя встроенные в Laravel функции [шифрования](#). Кроме того, преобразование `encrypted:array`, `encrypted:collection`, `encrypted:object`, `AsEncryptedArrayObject` и `AsEncryptedCollection` работает так же, как и их незашифрованные копии; однако, как и следовало ожидать, базовое значение зашифровано при хранении в вашей базе данных.

Поскольку окончательная длина зашифрованного текста непредсказуема и больше, чем его копия в виде обычного текста, убедитесь, что связанный столбец базы данных имеет тип `TEXT` или больше. Кроме того, поскольку значения зашифрованы в базе данных, вы не сможете запрашивать или искать зашифрованные значения атрибутов.

## Ротация ключей

## Типизация во время запроса

Иногда может потребоваться применить типизацию при выполнении запроса, например, при выборе сырого значения из таблицы. Например, рассмотрим следующий запрос:

```
use App\Models\Post;
use App\Models\User;

$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
        ->whereColumn('user_id', 'users.id')
])->get();
```

Атрибут `last_posted_at` результатов этого запроса будет простой строкой. Было бы замечательно, если бы мы могли применить типизацию `datetime` этого атрибута при выполнении запроса. К счастью, мы можем добиться этого с помощью метода `withCasts`:

```
$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
        ->whereColumn('user_id', 'users.id')
])->withCasts([
    'last_posted_at' => 'datetime'
])->get();
```

## # Пользовательская типизация

В Laravel есть множество встроенных полезных преобразователей; однако иногда требуется определить свои собственные. Для создания типа приведения выполните команду Artisan `make:cast`. Новый класс приведения будет размещен в вашем каталоге `app/Casts`:

Все пользовательские классы приведения должны реализовывать интерфейс `CastsAttributes`. Классы, реализующие этот интерфейс, должны определять методы `get` и `set`. Метод `get` отвечает за преобразование “сырого” значения из базы данных к типизированному значению, а метод `set` – должен преобразовывать типизированное значение в “сырое” значение, которое можно сохранить в базе

данных. В качестве примера мы повторно реализуем встроенный преобразователь `json` как пользовательский типизатор:

```
<?php

namespace App\Cast;

use Illuminate\Contracts\Database\Eloquent\CastAttributes;

class Json implements CastAttributes
{
    /**
     * Привести значение к пользовательскому типу.
     *
     * @param array<string, mixed> $attributes
     * @return array<string, mixed>
     */
    public function get(Model $model, string $key, mixed $value, array $attributes): array
    {
        return json_decode($value, true);
    }

    /**
     * Подготовить переданное значение к сохранению.
     *
     * @param array<string, mixed> $attributes
     */
    public function set(Model $model, string $key, mixed $value, array $attributes): array
    {
        return json_encode($value);
    }
}
```

После того как вы определили собственный типизатор, вы можете добавить его к атрибуту модели, используя его имя класса:

```
<?php

namespace App\Models;

use App\Cast\Json;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
```

```

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [
        'options' => Json::class,
    ];
}

```

## Типизация объект-значение

Вы не ограничены приведением значений к примитивным типам. Вы также можете преобразовать значения к объектам. Определение пользовательских типизаторов, которые преобразуют значения в объекты, очень похоже на приведение к примитивным типам; однако метод `set` должен возвращать массив пар ключ / значение, который будет использоваться для установки сырых значений, сохраняемых в модели.

В качестве примера мы определим собственный класс типизатора, который преобразует несколько значений модели в один объект-значение `Address`.

Предположим, что значение `Address` имеет два общедоступных свойства: `lineOne` и `lineTwo`:

```

<?php

namespace App\Cast;

use App\ValueObjects\Address as AddressValueObject;
use Illuminate\Contracts\Database\Eloquent\CastAttributes;
use Illuminate\Database\Eloquent\Model;
use InvalidArgumentException;

class Address implements CastAttributes
{
    /**
     * Преобразовать значение к пользовательскому типу.
     *
     * @param array<string, mixed> $attributes
     */
    public function get(Model $model, string $key, mixed $value, array $attributes):

```

```

{
    return new AddressValueObject(
        $attributes['address_line_one'],
        $attributes['address_line_two']
    );
}

/**
 * Подготовить переданное значение к сохранению.
 *
 * @param array<string, mixed> $attributes
 * @return array<string, string>
 */
public function set(Model $model, string $key, mixed $value, array $attributes):
{
    if (! $value instanceof AddressValueObject) {
        throw new InvalidArgumentException('The given value is not an Address in:
    }

    return [
        'address_line_one' => $value->lineOne,
        'address_line_two' => $value->lineTwo,
    ];
}
}

```

При приведении к объектам-значениям любые изменения, внесенные в объект-значения, будут автоматически синхронизированы с моделью до ее сохранения:

```

use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Value';

$user->save();

```

Если вы планируете сериализовать свои модели Eloquent, содержащие объекты-значения, в JSON или массивы, вам следует реализовать интерфейсы

`Illuminate\Contracts\Support\Arrayable` и  
`JsonSerializable` для объекта-значения.

## Кэширование объектов-значений

Когда атрибуты, которые приведены к объектам значений, вычислены, Eloquent кэширует их. Таким образом, при повторном доступе к атрибуту будет возвращен тот же самый экземпляр объекта.

Если вы хотите отключить поведение кэширования объектов в вашем пользовательском классе приведения, объягите `public` свойство `withoutObjectCaching` в вашем пользовательском классе приведения:

```
class Address implements CastsAttributes
{
    public bool $withoutObjectCaching = true;

    // ...
}
```

## Сериализация в массив и JSON

Когда модель Eloquent преобразуется в массив или JSON с использованием методов `toArray` и `toJson`, ваши пользовательские типизаторы объекты-значения обычно будут сериализованы, в частности, пока они (типоваторы) реализуют интерфейсы `Illuminate\Contracts\Support\Arrayable` и `JsonSerializable`. Однако при использовании объектов-значений, предоставляемых сторонними библиотеками, у вас может не быть возможности добавить эти интерфейсы к объекту.

Поэтому вы можете указать, что ваш собственный класс типизатора будет отвечать за сериализацию объекта-значения. Для этого ваш собственный класс типизатора должен реализовывать интерфейс `Illuminate\Contracts\Database\Eloquent\SerializesCastableAttributes`. В этом интерфейсе указано, что ваш класс должен содержать метод `serialize`, возвращающий сериализованную форму вашего объекта значения:

```
/**
 * Получить сериализованное представление значения.
 */
```

```
* @param array<string, mixed> $attributes
*/
public function serialize(Model $model, string $key, mixed $value, array $attributes)
{
    return (string) $value;
}
```

## Входящая типизация

Иногда требуется написать свой типизатор, который только преобразует указанные значения атрибутов модели, и не выполняет никаких операций при обращении к этим атрибутам.

Пользовательские типизаторы только для входящих значений должны реализовывать интерфейс [CastsInboundAttributes](#), требующий определение метода `set`. Вызовите команду Artisan `make:cast` с опцией `--inbound`, чтобы сгенерировать класс приведения только для входящих значений:

```
php artisan make:cast Hash --inbound
```

Классическим примером только входящей типизации является «хеширование». Например, мы можем определить типизатор, которое хеширует входящие значения с использованием указанного алгоритма:

```
<?php

namespace App\Cast;

use Illuminate\Contracts\Database\Eloquent\CastsInboundAttributes;
use Illuminate\Database\Eloquent\Model;

class Hash implements CastsInboundAttributes
{

    /**
     * Создать новый экземпляр класса типизации.
     */
    public function __construct(
        protected string|null $algorithm = null,
    ) {}
```

```

/**
 * Подготовить переданное значение к сохранению.
 *
 * @param array<string, mixed> $attributes
 */
public function set(Model $model, string $key, mixed $value, array $attributes):
{
    return is_null($this->algorithm)
        ? bcrypt($value)
        : hash($this->algorithm, $value);
}
}

```

## Параметры типизации

При добавлении пользовательского типизатора к модели, параметры типизатора задаются отделением их от имени класса с помощью символа `:` и разделением нескольких параметров запятыми. Параметры будут переданы в конструктор класса типизатора:

```

/**
 * Получение атрибутов, которые должны быть типизированы.
 *
 * @return array<string, string>
 */
protected function casts(): array
{
    return [
        'secret' => Hash::class.':sha256',
    ];
}

```

## Интерфейс Castable

Вы можете разрешить объектам-значениям вашего приложения определять свои собственные классы типизаторы. Вместо указания пользовательской типизации в модели, вы можете альтернативно указать класс, который реализует интерфейс [Illuminate\Contracts\Database\Eloquent\Castable](#):

```

use App\ValueObjects\Address;

protected function casts(): array

```

```
{  
    return [  
        'address' => Address::class,  
    ];  
}
```

Объекты, реализующие интерфейс `Castable`, должны определять метод `castUsing`, который возвращает имя [пользовательского класса типизатора](#), отвечающего за двустороннее преобразование:

```
<?php  
  
namespace App\ValueObjects;  
  
use Illuminate\Contracts\Database\Eloquent\Castable;  
use App\Cast\Address as AddressCast;  
  
class Address implements Castable  
{  
    /**  
     * Получить имя класса типизатора для использования двустороннего преобразования  
     *  
     * @param array<string, mixed> $arguments  
     */  
    public static function castUsing(array $arguments): string  
    {  
        return AddressCast::class;  
    }  
}
```

При использовании классов `Castable` вы все равно можете указывать аргументы в методе `casts`. Аргументы будут переданы методу `castUsing`:

```
use App\ValueObjects\Address;  
  
protected function casts(): array  
{  
    return [  
        'address' => Address::class.':argument',  
    ];  
}
```

## Интерфейс Castable и анонимные классы типизаторов

Комбинируя `castable` и [анонимными классами](#) PHP, вы можете определить объект-значение и его логику преобразования как единый типизируемый объект. Для этого верните анонимный класс из метода `castUsing` вашего объекта-значения.

Анонимный класс должен реализовывать интерфейс `CastsAttributes`:

```
<?php

namespace App\ValueObjects;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class Address implements Castable
{
    // ...

    /**
     * Получить имя класса типизатора для использования двустороннего преобразования
     *
     * @param array<string, mixed> $arguments
     */
    public static function castUsing(array $arguments): CastsAttributes
    {
        return new class implements CastsAttributes
        {
            public function get(Model $model, string $key, mixed $value, array $attr):
            {
                return new Address(
                    $attributes['address_line_one'],
                    $attributes['address_line_two']
                );
            }

            public function set(Model $model, string $key, mixed $value, array $attr):
            {
                return [
                    'address_line_one' => $value->lineOne,
                    'address_line_two' => $value->lineTwo,
                ];
            }
        };
    }
}
```



# Eloquent · Ресурсы API (Resource)

- # Введение
- # Генерация ресурсов
- # Обзор концепции
  - # Коллекции ресурса
- # Написание ресурсов
  - # Обертывание данных
  - # Постстраничная разбивка
  - # Условные атрибуты
  - # Условные отношения
  - # Добавление метаданных
- # Ответы ресурса

## # Введение

При создании API вам может потребоваться слой преобразования, находящийся между вашими моделями Eloquent и ответами JSON, которые фактически возвращаются пользователям вашего приложения. Например, бывает необходимо отображать определенные атрибуты только для некоторого сегмента пользователей, а не для всех, или бывает необходимо всегда отображать определенные отношения в JSON-представление ваших моделей. Классы ресурсов Eloquent позволяют легко и выразительно преобразовывать модели и коллекции моделей в JSON.

Конечно, вы всегда можете преобразовать модели или коллекции Eloquent в JSON, используя их методы `toJson`; однако ресурсы Eloquent обеспечивают более детальный и надежный контроль над сериализацией в JSON ваших моделей и их отношений.

## # Генерация ресурсов

Ресурсы расширяют класс `Illuminate\Http\Resources\Json\JsonResource`. Чтобы сгенерировать новый ресурс, используйте команду `make:resource Artisan`. Эта команда поместит новый класс ресурса в каталог `app/Http/Resources` вашего приложения:

```
php artisan make:resource UserResource
```

## Генерация коллекций ресурса

Помимо создания ресурсов, преобразующих отдельные модели, вы можете создавать ресурсы, отвечающие за преобразование коллекций моделей. Это позволяет вашим ответам JSON включать ссылки и другую метаинформацию, имеющую отношение ко всей коллекции конкретного ресурса.

Чтобы сгенерировать новую коллекцию ресурса, вы должны использовать флаг `--collection` при создании ресурса. Или включение слова `Collection` в имя ресурса укажет Laravel, что он должен создать коллекцию ресурса. Коллекции ресурса расширяют класс `Illuminate\Http\Resources\Json\ResourceCollection`:

```
php artisan make:resource User --collection
```

```
php artisan make:resource UserCollection
```

## # Обзор концепции

Это лишь общий обзор ресурсов и коллекций ресурса. Мы настоятельно рекомендуем вам прочитать другие разделы этой документации, чтобы получить более глубокое понимание возможностей создания и настройки ресурса, предлагаемые вам.

Прежде чем углубляться во все варианты, доступные вам при написании ресурсов, давайте сначала рассмотрим, как ресурсы используются в Laravel. Класс ресурсов

представляет собой единую модель, которую необходимо преобразовать в структуру JSON. Например, вот простой класс ресурса [UserResource](#):

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Преобразовать ресурс в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Каждый класс ресурсов определяет метод [toArray](#), возвращающий массив атрибутов, которые должны быть преобразованы в JSON, когда ресурс возвращается в качестве ответа из метода маршрута или контроллера.

Обратите внимание, что мы можем получить доступ к свойствам модели непосредственно из переменной [\\$this](#). Это связано с тем, что класс ресурсов автоматически проксирует свойства и методы к базовой модели для удобства доступа. Как только ресурс определен, он может быть возвращен из маршрута или контроллера. Ресурс принимает основной экземпляр модели через свой конструктор:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
```

```
        return new UserResource(User::findOrFail($id));
    });
}
```

## Коллекции ресурса

Если вы возвращаете коллекцию ресурса или ответ с постраничной разбивкой, то вы должны использовать метод `collection` класса ресурса, при создании экземпляра ресурса в вашем маршруте или контроллере:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Обратите внимание, что это не позволит добавить пользовательские метаданные, которые могут потребоваться при возвращении с вашей коллекцией. Если вы хотите получить больший контроль над ответом коллекции ресурса, то вы можете создать выделенный ресурс для представления коллекции:

```
php artisan make:resource UserCollection
```

После создания класса коллекции ресурса, вы можете легко определить любые метаданные, которые должны быть включены в ответ:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Преобразовать коллекцию ресурса в массив.
     *
     * @return array<int|string, mixed>
     */
    public function toArray(Request $request): array
```

```
{  
    return [  
        'data' => $this->collection,  
        'links' => [  
            'self' => 'link-value',  
        ],  
    ];  
}  
}  
}
```

После определения вашей коллекции ресурса, ее можно вернуть из маршрута или контроллера:

```
use App\Http\Resources\UserCollection;  
use App\Models\User;  
  
Route::get('/users', function () {  
    return new UserCollection(User::all());  
});
```

## Сохранение ключей коллекции

При возврате коллекции ресурсов из маршрута, Laravel сбрасывает ключи коллекции для расположения их в числовом порядке. Однако, вы можете добавить свойство `$preserveKeys` в свой класс ресурса, указывающее, должны ли сохраняться исходные ключи коллекции:

```
<?php  
  
namespace App\Http\Resources;  
  
use Illuminate\Http\Resources\Json\JsonResource;  
  
class UserResource extends JsonResource  
{  
    /**  
     * Указывает, следует ли сохранить ключи коллекции ресурса.  
     *  
     * @var bool  
     */  
    public $preserveKeys = true;  
}
```

Когда для свойства `$preserveKeys` установлено значение `true`, ключи коллекции будут сохранены, когда коллекция будет возвращена из маршрута или контроллера:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

## Настройка базового класса ресурсов

Обычно свойство `$this->collection` коллекции ресурса автоматически заполняется результатом сопоставления каждого элемента коллекции с его единственным классом ресурсов. Предполагается, что единственным классом ресурса является имя коллекции без завершающей части `Collection`. Кроме того, в зависимости от личных предпочтений, класс ресурсов в единственном числе может иметь суффикс `Resource`, а может и не иметь его.

Например, `UserCollection` попытается сопоставить переданные экземпляры пользователя с ресурсом `UserResource`. Чтобы изменить это поведение, вы можете переопределить свойство `$collects` вашей коллекции ресурса:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Ресурс, используемый при формировании коллекции.
     *
     * @var string
     */
    public $collects = Member::class;
}
```

## # Написание ресурсов

Если вы не читали [обзор концепции](#), настоятельно рекомендуется сделать это, прежде чем приступить к работе с этой документацией.

Ресурсам нужно только преобразовать данную модель в массив. Итак, каждый ресурс содержит метод `toArray`, переводящий атрибуты вашей модели в удобный для API массив, который может быть возвращен из маршрутов или контроллеров вашего приложения:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Преобразовать ресурс в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Как только ресурс определен, он может быть возвращен непосредственно из маршрута или контроллера:

```
use App\Http\Resources\UserResource;
use App\Models\User;
```

```
Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});
```

## Отношения

Если вы хотите включить связанные ресурсы в свой ответ, вы можете добавить их в массив, возвращаемый методом вашего ресурса `toArray`. В этом примере мы будем использовать метод `collection` ресурса `PostResource`, чтобы добавить посты пользователя из блога в ответ ресурса:

```
use App\Http\Resources\PostResource;
use Illuminate\Http\Request;

/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

{Если вы хотите включить отношения только тогда, когда они уже загружены, ознакомьтесь с документацией по [условным отношениям](#).

## Коллекции ресурса

В то время как ресурсы преобразуют одну модель в массив, коллекции ресурса преобразуют коллекцию моделей в массив. Однако, необязательно определять класс коллекции ресурса для каждой из ваших моделей, поскольку все ресурсы

предоставляют метод `collection` для генерации «специальной» (ad hoc) коллекции ресурсов на лету:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

Однако, если вам нужно настроить метаданные, возвращаемые с коллекцией, необходимо определить собственную коллекцию ресурса:

<?php

```
namespace App\Http\Resources;

use Illuminate\Http\Request
use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Преобразовать коллекцию ресурса в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Как и отдельные ресурсы, коллекции ресурса могут быть возвращены непосредственно из маршрутов или контроллеров:

```
use App\Http\Resources\UserCollection;
use App\Models\User;
```

```
Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

## Обертывание данных

По умолчанию, ваш самый верхний ресурс будет заключен в ключ `data`, когда ответ ресурса преобразуется в JSON. Так, например, типичный ответ коллекции ресурса выглядит следующим образом:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ]
}
```

Если вы хотите отключить обертывание самого верхнего ресурса, то вы должны вызвать метод `withoutWrapping` базового класса `Illuminate\Http\Resources\Json\JsonResource`. Обычно вы должны вызывать этот метод из вашего `AppServiceProvider` или другого [сервис-провайдера](#):

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Регистрация любых служб приложения.
     *
     * @return void
    
```

```
 */
public function register(): void
{
    // ...
}

/**
 * Загрузка любых служб приложения.
 *
 * @return void
 */
public function boot(): void
{
    JsonResource::withoutWrapping();
}

}
```

Метод `withoutWrapping` влияет только на самый верхний уровень ответа и не удаляет ключи `data`, которые вы вручную добавляете в свои собственные коллекции ресурса.

## Обертывание вложенных ресурсов

У вас есть полная свобода определять, как обернуты отношения между вашими ресурсами. Если вы хотите, чтобы все коллекции ресурсов были обернуты ключом `data`, независимо от их вложенности, то вы должны определить класс коллекции для каждого ресурса и вернуть коллекцию с ключом `data`.

Вам может быть интересно: не приведет ли это к тому, что верхний уровень вашего ресурса будет дважды обернут ключом `data`? Не волнуйтесь, Laravel не позволит вашим ресурсам быть случайно обернутыми двойной оберткой, поэтому вам не нужно беспокоиться об уровне вложенности трансформируемой коллекции ресурсов:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;
```

```

class CommentsCollection extends ResourceCollection
{
    /**
     * Преобразовать коллекцию ресурса в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return ['data' => $this->collection];
    }
}

```

## Обертывание данных и постраничная разбивка

При возврате разбитых на страницы коллекций через ответ ресурса, Laravel обернет ваши данные ресурса в ключ `data`, даже если был вызван метод `withoutWrapping`. Это потому, что разбитые на страницы ответы всегда содержат ключи `meta` и `links` с информацией о состоянии постраничной разбивки:

```

{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links": {
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=1",
        "prev": null,
        "next": null
    },
    "meta": {
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/users",
        "per_page": 15,
    }
}

```

```
        "to": 10,
        "total": 10
    }
}
```

## Постраничная разбивка

Вы можете передать экземпляр пагинатора Laravel методу `collection` ресурса или вашей коллекции ресурса:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Ответы с постраничной разбивкой всегда содержат ключи `meta` и `links` с информацией о состоянии пагинатора:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links":{
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/users",
```

```
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

## Настройка информации о постраничной навигации

Если вы хотите настроить информацию, включаемую в ключи `links` или `meta` ответа пагинации, вы можете определить метод `paginationInformation` в ресурсе. Этот метод получит данные `$paginated` и массив `$default` информации, который содержит ключи `links` и `meta`:

```
/**
 * Настройка информации о постраничной навигации для ресурса.
 *
 * @param \Illuminate\Http\Request $request
 * @param array $paginated
 * @param array $default
 * @return array
 */
public function paginationInformation($request, $paginated, $default)
{
    $default['links']['custom'] = 'https://example.com';

    return $default;
}
```

## Условные атрибуты

По желанию можно включить атрибут в ответ ресурса, только если какое-то условие выполнено. Например, бывает необходимо включить значение, только если текущий пользователь является «администратором». Laravel предлагает множество вспомогательных методов, которые помогут вам в этой ситуации. Метод `when` используется для условного добавления атрибута в ответ ресурса:

```
/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
```

```
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        'secret' => $this->when($request->user()->isAdmin(), 'secret-value'),  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}
```

В этом примере ключ `secret` будет возвращен в конечном ответе ресурса только в том случае, если метод `isAdmin` аутентифицированного пользователя вернет `true`. Если метод возвращает `false`, то ключ `secret` будет удален из ответа ресурса перед его отправкой клиенту. Метод `when` позволяет вам выразительно определять ваши ресурсы, не прибегая к условным операторам при построении массива.

Метод `when` также принимает замыкание в качестве второго аргумента, позволяя вам вычислить результирующее значение, только если переданное условие истинно:

```
'secret' => $this->when($request->user()->isAdmin(), function () {  
    return 'secret-value';  
}),
```

Метод `whenHas` может быть использован для включения атрибута, если он действительно присутствует в основной модели:

```
'name' => $this->whenHas('name'),
```

Кроме того, метод `whenNotNull` может быть использован для включения атрибута в ответ ресурса, если атрибут не равен null:

```
'name' => $this->whenNotNull($this->name),
```

## Слияние условных атрибутов

Иногда у вас может быть несколько атрибутов, которые следует включать в ответ ресурса только при одном и том же условии. В этом случае вы можете

использовать метод `mergeWhen` для включения атрибутов в ответ только в том случае, если переданное условие истинно:

```
/**  
 * Преобразовать ресурс в массив.  
 *  
 * @return array<string, mixed>  
 */  
public function toArray(Request $request): array  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        $this->mergeWhen($request->user()->isAdmin(), [  
            'first-secret' => 'value',  
            'second-secret' => 'value',  
        ]),  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}
```

Опять же, если переданное условие равносильно `false`, то эти атрибуты будут удалены из ответа ресурса перед его отправкой клиенту.

Метод `mergeWhen` не следует использовать в массивах, в которых смешиваются строковые и числовые ключи. Кроме того, его не следует использовать в массивах с цифровыми ключами, которые не упорядочены последовательно.

## Условные отношения

В дополнение к условной загрузке атрибутов, вы можете условно включать отношения в свои ответы ресурса в зависимости от того, было ли отношение уже загружено в модель. Это позволяет вашему контроллеру решать, какие отношения должны быть загружены в модель, и ваш ресурс может легко включить их, только когда они действительно были загружены. В конечном итоге это позволяет избежать проблем «N+1» с запросами в ваших ресурсах.

Метод `whenLoaded` используется для условной загрузки отношения. Чтобы избежать ненужной загрузки отношений, этот метод принимает имя отношения вместо самого отношения:

```
use App\Http\Resources\PostResource;

/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

В этом примере, если отношение не было загружено, ключ `posts` будет удален из ответа ресурса перед его отправкой клиенту.

## Условные подсчеты отношений

Помимо условного включения отношений, вы также можете условно включать "счетчики" отношений в ответах вашего ресурса в зависимости от того, был ли загружен счетчик отношений в модели:

```
new UserResource($user->loadCount('posts'));
```

Метод `whenCounted` может быть использован для условного включения подсчета отношения в ответ вашего ресурса. Этот метод избегает лишнего включения атрибута, если подсчет отношения отсутствует:

```
/**
 * Transform the resource into an array.
 *
 * @return array<string, mixed>
```

```

*/
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts_count' => $this->whenCounted('posts'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}

```

В данном примере, если подсчет отношения posts не был загружен, ключ `posts_count` будет удален из ответа ресурса перед его отправкой клиенту.

Другие типы агрегатов, такие как `avg`, `sum`, `min` и `max`, также могут быть условно загружены с использованием метода `whenAggregated`:

```

'words_avg' => $this->whenAggregated('posts', 'words', 'avg'),
'words_sum' => $this->whenAggregated('posts', 'words', 'sum'),
'words_min' => $this->whenAggregated('posts', 'words', 'min'),
'words_max' => $this->whenAggregated('posts', 'words', 'max'),

```

## Условная сводная информация

В дополнение к условному включению информации об отношениях в ответах ваших ресурсов, вы можете условно включать данные из сводных таблиц отношений «многие ко многим» с помощью метода `whenPivotLoaded`. Метод `whenPivotLoaded` принимает имя сводной таблицы в качестве своего первого аргумента. Второй аргумент должен быть замыканием, возвращающим значение, если в модели доступна сводная информация:

```

/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,

```

```
'expires_at' => $this->whenPivotLoaded('role_user', function () {
    return $this->pivot->expires_at;
}),
],
}
```

Если ваши отношения используют [пользовательскую модель сводной таблицы](#), то вы можете передать экземпляр модели сводной таблицы в качестве первого аргумента методу `whenPivotLoaded`:

```
'expires_at' => $this->whenPivotLoaded(new Membership, function () {
    return $this->pivot->expires_at;
}),
```

Если ваша сводная таблица использует аксессор, отличный от `pivot`, то вы можете использовать метод `whenPivotLoadedAs`:

```
/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function () {
            return $this->subscription->expires_at;
}),
    ];
}
```

## Добавление метаданных

Некоторые стандарты API JSON требуют добавления метаданных в ответы ваших ресурсов и коллекции ресурсов. Это часто включает такие вещи, как «ссылки» на ресурс или связанные ресурсы, или метаданные о самом ресурсе. Если вам нужно вернуть дополнительные метаданные о ресурсе, включите их в свой метод `toArray`. Например, вы можете включить информацию `links` при преобразовании коллекции ресурса:

```
/**
 * Преобразовать ресурс в массив.
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

При возврате дополнительных метаданных из ваших ресурсов вам никогда не придется беспокоиться о случайном переопределении ключей `links` или `meta`, которые автоматически добавляются Laravel при возврате ответов с постраничной разбивкой. Любые дополнительные `links`, которые вы определяете, будут объединены с предоставленными пагинатором.

## Метаданные верхнего уровня

По желанию можно включить в ответ ресурса только определенные метаданные, если ресурс является самым верхним из возвращаемых ресурсов. Обычно это метаинформация об ответе в целом. Чтобы определить эти метаданные, добавьте метод `with` к вашему классу ресурсов. Этот метод должен возвращать массив метаданных, которые будут включены в ответ ресурса, только если ресурс является самым верхним ресурсом, который преобразуется:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Преобразовать коллекцию ресурса в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
```

```

{
    return parent::toArray($request);
}

/**
 * Получить дополнительные данные, возвращаемые с массивом ресурса.
 *
 * @return array<string, mixed>
 */
public function with(Request $request): array
{
    return [
        'meta' => [
            'key' => 'value',
        ],
    ];
}
}

```

## Добавление метаданных при создании ресурсов

Вы также можете добавить данные верхнего уровня при создании экземпляров ресурсов в своем маршруте или контроллере. Метод `additional`, доступный для всех ресурсов, принимает массив данных, которые должны быть добавлены в ответ ресурса:

```

return (new UserCollection(User::all()->load('roles')))
    ->additional(['meta' => [
        'key' => 'value',
    ]]);

```

## # Ответы ресурса

Как вы уже читали, ресурсы могут быть возвращены напрямую из маршрутов и контроллеров:

```

use App\Http\Resources\User as UserResource;
use App\Models\User;

Route::get('/user/{id}', function (string $id) {
    return new UserResource(User::findOrFail($id));
});

```

Иногда требуется настроить исходящий HTTP-ответ перед его отправкой клиенту. Это можно сделать двумя способами. Во-первых, вы можете связать метод `response` с ресурсом. Этот метод вернет экземпляр `Illuminate\Http\JsonResponse`, что даст вам полный контроль над заголовками ответа:

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
        ->response()
        ->header('X-Value', 'True');
});
```

В качестве альтернативы вы можете определить метод `withResponse` внутри самого ресурса. Этот метод будет вызываться, только когда ресурс будет возвращен как самый верхний ресурс в ответе:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * Преобразовать ресурс в массив.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Настроить исходящий ответ для ресурса.
     */
    public function withResponse(Request $request, JsonResponse $response): void
    {
```

```
$response->header('X-Value', 'True');  
}  
}
```

# Eloquent · Сериализация

- # Введение
- # Сериализация моделей и коллекций
  - # Сериализация в массивы
  - # Сериализация в JSON
- # Скрытие атрибутов из JSON
- # Добавление значений в JSON
- # Сериализация Даты

## # Введение

При создании API-интерфейсов с использованием Laravel вам часто нужно преобразовывать свои модели и отношения в массивы или JSON. Eloquent включает удобные методы для выполнения этих преобразований, а также для управления тем, какие атрибуты включаются в сериализованное представление ваших моделей.

Чтобы получить еще более надежный способ обработки JSON-сериализации модели Eloquent и коллекции, ознакомьтесь с документацией на [Ресурсы API Eloquent](#).

## # Сериализация моделей и коллекций

### Сериализация в массивы

Чтобы преобразовать модель и ее загруженные [отношения](#) в массив, вы должны использовать метод `toArray`. Этот метод является рекурсивным, поэтому все

атрибуты и все отношения (включая отношения отношений) будут преобразованы в массивы:

```
use App\Models\User;

$user = User::with('roles')->first();

return $user->toArray();
```

Метод `attributesToArray` используется для преобразования атрибутов модели в массив, но не его отношений:

```
$user = User::first();

return $user->attributesToArray();
```

Вы также можете преобразовать целые [коллекции](#) моделей в массивы, вызвав метод `toArray` экземпляра коллекции:

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

return $users->toArray();
```

## Сериализация в JSON

Чтобы преобразовать модель в JSON, вы должны использовать метод `toJson`. Как и `toArray`, метод `toJson` является рекурсивным, поэтому все атрибуты и отношения будут преобразованы в JSON. Вы также можете указать любые параметры кодировки JSON, которые [поддерживаются PHP](#):

```
use App\Models\User;

$user = User::find(1);

return $user->toJson();

return $user->toJson(JSON_PRETTY_PRINT);
```

В качестве альтернативы вы можете преобразовать модель или коллекцию в строку, которая автоматически вызовет метод `toJson` модели или коллекции:

```
return (string) User::find(1);
```

Поскольку модели и коллекции преобразуются в JSON при преобразовании в строку, вы можете возвращать объекты Eloquent непосредственно из маршрутов или контроллеров вашего приложения. Laravel автоматически сериализует ваши модели и коллекции Eloquent в JSON, когда они возвращаются из маршрутов или контроллеров:

```
Route::get('/users', function () {
    return User::all();
});
```

## Отношения

Когда модель Eloquent преобразуется в JSON, ее загруженные отношения автоматически включаются в качестве атрибутов в объект JSON. Кроме того, хотя методы-отношения Eloquent определены с использованием имен методов в «верблюжьей нотации», атрибут JSON отношения будет в «змеиной нотации».

## # Скрытие атрибутов из JSON

По желанию можно исключить атрибуты, такие как пароли, содержащиеся в массиве модели или представлении JSON. Для этого добавьте в модель свойство `$hidden`. Атрибуты, перечисленные в массиве свойств `$hidden`, не будут включены в сериализованное представление модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Атрибуты, которые должны быть скрыты из массивов.

```

```
*  
 * @var array  
 */  
protected $hidden = ['password'];  
}
```

Чтобы скрыть отношения, добавьте имя метода-отношения к свойству `$hidden` модели Eloquent.

В качестве альтернативы вы можете использовать свойство `visible` для определения «разрешенного списка» атрибутов, которые должны быть включены в массив модели и представление JSON. Все атрибуты, отсутствующие в массиве `$visible`, будут скрыты при преобразовании модели в массив или JSON:

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model  
{  
    /**  
     * Атрибуты, которые должны быть видны в массивах.  
     *  
     * @var array  
     */  
protected $visible = ['first_name', 'last_name'];  
}
```

## Временное изменение видимости атрибута

По желанию можно сделать некоторые обычно скрытые атрибуты видимыми на конкретном экземпляре модели, для этого используйте метод `makeVisible`. Метод `makeVisible` возвращает экземпляр модели:

```
return $user->makeVisible('attribute')->toArray();
```

Аналогично, если вы хотите скрыть некоторые атрибуты, которые обычно видны, вы можете использовать метод `makeHidden`:

```
return $user->makeHidden('attribute')->toArray();
```

Если вам нужно временно переопределить все видимые или скрытые атрибуты, вы можете использовать методы `setVisible` и  `setHidden` соответственно:

```
return $user->setVisible(['id', 'name'])->toArray();
```

```
return $user->setHidden(['email', 'password', 'remember_token'])->toArray();
```

## # Добавление значений в JSON

Иногда при преобразовании моделей в массивы или JSON вы можете добавить атрибуты, которым нет соответствующего столбца в вашей базе данных. Для этого сначала определите [аксессоры](#) для значения:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Определить, является ли пользователь администратором.
     */
    protected function isAdmin(): Attribute
    {
        return new Attribute(
            get: fn () => 'yes',
        );
    }
}
```

Если вы хотите, чтобы аксессор всегда добавлялся к массиву и JSON-представлению вашей модели, добавьте имя атрибута к свойству `appends` модели.

Обратите внимание, что на имена атрибутов обычно ссылаются в «змеиной нотации», даже если аксессор определяется с помощью «верблюжьей нотации»:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Аксессоры, добавляемые к массиву модели.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

После того как атрибут был добавлен в список `appends`, он будет включен как в массив модели, так и в представления JSON. Атрибуты в массиве `appends` также будут учитывать настройки `visible` и `hidden`, заданные в модели.

## Добавление во время запроса

Во время выполнения скрипта вы можете указать экземпляру модели добавить дополнительные атрибуты с помощью метода `append`. Или вы можете использовать метод `setAppends`, чтобы переопределить весь массив добавленных свойств для конкретного экземпляра модели:

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

## # Сериализация Даты

### Настройка формата даты по умолчанию

Вы можете настроить формат сериализации по умолчанию для всех дат вашей модели, переопределив метод `serializeDate` вашей модели. Этот метод не влияет на

форматирование дат для их сохранения в базе данных:

```
/**  
 * Подготовить дату для сериализации массива / JSON.  
 */  
protected function serializeDate(DateTimeInterface $date): string  
{  
    return $date->format('Y-m-d');  
}
```

## Настройка формата даты для каждого атрибута

Вы можете настроить формат сериализации отдельных атрибутов даты, указав формат даты при [объявлении типизации](#) модели:

```
protected function casts(): array  
{  
    return [  
        'birthday' => 'date:Y-m-d',  
        'joined_at' => 'datetime:Y-m-d H:00',  
    ];  
}
```

# Eloquent: Фабрики (Factory)

# Введение

# Определение фабрик моделей

# Генерация фабрик

# Состояния фабрик

# Хуки фабрик

# Создание моделей с использованием фабрик

# Инициализация экземпляров моделей

# Сохранение моделей

# Последовательность состояний (Sequences)

# Отношения

# Отношения Has Many

# Отношения Belongs To

# Отношения Many To Many

# Полиморфные отношения

# Определение отношений внутри фабрик

# Повторное использование существующей модели для отношений

## # Введение

При тестировании вашего приложения вам может потребоваться вставить несколько записей в вашу базу данных. Вместо того чтобы вручную указывать значение каждого столбца, Laravel позволяет вам определять набор атрибутов по умолчанию для каждой из ваших [моделей Eloquent](#), используя фабрики моделей.

Чтобы увидеть пример написания фабрики, взгляните на файл `database/factories/UserFactory.php` в вашем приложении. Эта фабрика включена во все новые приложения Laravel и содержит следующее определение фабрики:

```
namespace Database\Factories;
```

```

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\User>
 */
class UserFactory extends Factory
{
    /**
     * The current password being used by the factory.
     */
    protected static ?string $password;

    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => static::$password ??= Hash::make('password'),
            'remember_token' => Str::random(10),
        ];
    }

    /**
     * Indicate that the model's email address should be unverified.
     */
    public function unverified(): static
    {
        return $this->state(fn (array $attributes) => [
            'email_verified_at' => null,
        ]);
    }
}

```

Как видите, фабрики – это классы, которые расширяют базовый класс фабрики Laravel и определяют метод `definition`. Метод `definition` возвращает набор значений атрибутов по умолчанию, которые должны применяться при создании модели с использованием фабрики.

С помощью помощник `fake` фабрики имеют доступ к библиотеке PHP [Faker](#), которая позволяет удобно генерировать различные виды случайных данных для тестирования и заполнения базы данных.

Вы можете изменить языковой стандарт Faker для своего приложения, обновив параметр `faker_locale` в файле конфигурации `config/app.php`.

## # Определение фабрик моделей

### Генерация фабрик

Чтобы сгенерировать новую фабрику, используйте команду `make:factory Artisan`:

```
php artisan make:factory PostFactory
```

Эта команда поместит новый класс фабрики в каталог `database/factories`.

### Соглашение для определения моделей и фабрик

После того как вы определили свои фабрики, вы можете использовать статический метод `factory` предоставляемый вашим моделям с помощью трейта `Illuminate\Database\Eloquent\Factories\HasFactory`, чтобы создать экземпляр фабрики для этой модели.

Метод `factory` трейта `HasFactory` будет использовать соглашения для определения подходящей фабрики для модели. В частности, метод будет искать фабрику в пространстве имен `Database\Factories`, имя класса которой соответствует имени модели и имеет суффикс `Factory`. Если эти соглашения не применимы к вашему конкретному приложению или фабрике, вы можете перезаписать метод `newFactory` вашей модели, чтобы напрямую возвращать экземпляр соответствующей фабрики модели:

```
use Database\Factories\Administration\FlightFactory;
```

```
/**  
 * Создать новый экземпляр фабрики для модели.  
 */  
protected static function newFactory()  
{  
    return FlightFactory::new();  
}
```

Затем определите свойство `model` в соответствующей фабрике:

```
use App\Administration\Flight;  
use Illuminate\Database\Eloquent\Factories\Factory;  
  
class FlightFactory extends Factory  
{  
    /**  
     * Название модели, соответствующей фабрике.  
     *  
     * @var class-string<\\Illuminate\\Database\\Eloquent\\Model>  
     */  
    protected $model = Flight::class;  
}
```

## Состояния фабрик

Методы управления состоянием позволяют вам определять дискретные изменения, которые могут быть применены к вашим фабрикам моделей в любой их комбинации. Например, ваша фабрика `Database\Factories\UserFactory` может содержать метод состояния `suspended`, который изменяет одно из значений атрибута по умолчанию.

Методы преобразования состояния обычно вызывают метод `state` базового класса фабрики Laravel. Метод `state` принимает замыкание, которое получит массив изначально определенных для фабрики атрибутов, и должен вернуть массив изменяемых атрибутов:

```
use Illuminate\Database\Eloquent\Factories\Factory;  
  
/**  
 * Указать, что аккаунт пользователя временно приостановлен.  
 */  
public function suspended(): Factory  
{
```

```
return $this->state(function (array $attributes) {
    return [
        'account_status' => 'suspended',
    ];
});
}
```

## "Trashed" State

Если ваша модель Eloquent поддерживает [программное удаление](#), вы можете вызвать встроенный метод состояния `trashed` чтобы указать, что созданная модель уже должна быть "программно удалена". Вам не нужно ручным образом определять состояние `trashed` так как оно автоматически доступно для всех фабрик:

```
use App\Models\User;

$user = User::factory()->trashed()->create();
```

## Хуки фабрик

Хуки фабрик регистрируются с использованием методов `afterMaking` и `afterCreating` и позволяют выполнять дополнительные задачи после инициализации или создания модели. Вы должны зарегистрировать эти хуки, переопределив метод `configure` в вашем классе фабрики. Этот метод будет автоматически вызываться Laravel при создании экземпляра фабрики:

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    /**
     * Конфигурация фабрики модели.
     */
    public function configure(): static
    {
        return $this->afterMaking(function (User $user) {
            // ...
        })->afterCreating(function (User $user) {
            // ...
        });
    }
}
```

```
// ...
});

}

// ...
}
```

Вы также можете зарегистрировать хуки фабрики внутри методов состояния для выполнения дополнительных задач, специфичных для определенного состояния:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * Указать, что аккаунт пользователя временно приостановлен.
 */
public function suspended(): Factory
{
    return $this->state(function (array $attributes) {
        return [
            'account_status' => 'suspended',
        ];
    })->afterMaking(function (User $user) {
        // ...
    })->afterCreating(function (User $user) {
        // ...
    });
}
```

## # Создание моделей с использованием фабрик

### Инициализация экземпляров моделей

После того как вы определили свои фабрики, вы можете использовать статический метод `factory`, предоставляемый вашим моделям с помощью трейта `Illuminate\Database\Eloquent\Factories\HasFactory`, чтобы инициализировать экземпляр фабрики для этой модели. Давайте посмотрим на несколько примеров создания моделей. Во-первых, мы воспользуемся методом `make` для создания моделей без сохранения в базе данных:

```
use App\Models\User;
```

```
$user = User::factory()->make();
```

Вы можете создать коллекцию из множества моделей, используя метод `count`:

```
$users = User::factory()->count(3)->make();
```

## Применение состояний

Вы также можете применить к моделям любое из ваших [состояний](#). Если вы хотите применить к моделям несколько изменений состояния, то вы можете просто вызвать методы преобразования состояния напрямую:

```
$users = User::factory()->count(5)->suspended()->make();
```

## Переопределение атрибутов

Если вы хотите переопределить некоторые значения по умолчанию для ваших моделей, вы можете передать массив значений методу `make`. Будут заменены только указанные атрибуты, в то время как для остальных атрибутов сохранятся значения по умолчанию, указанные в фабрике:

```
$user = User::factory()->make([
    'name' => 'Abigail Otwell',
]);
```

В качестве альтернативы, метод `state` может быть вызван непосредственно на экземпляре фабрики для выполнения быстрого преобразования состояния:

```
$user = User::factory()->state([
    'name' => 'Abigail Otwell',
])->make();
```

[Защита от массового назначения](#) автоматически отключается при создании моделей с

использованием фабрик..

## Сохранение моделей

Метод `create` инициализирует экземпляры модели и сохраняет их в базе данных с помощью метода `save` модели Eloquent:

```
use App\Models\User;

// Создаем один экземпляр `App\Models\User` ...
$user = User::factory()->create();

// Создаем три экземпляра `App\Models\User` .
$users = User::factory()->count(3)->create();
```

Вы можете переопределить атрибуты модели по умолчанию, передав массив атрибутов методу `create`:

```
$user = User::factory()->create([
    'name' => 'Abigail',
]);
```

## Последовательность состояний (Sequences)

Иногда может возникнуть желание чередовать значение определенного атрибута модели для каждой создаваемой модели. Вы можете добиться этого, определив последовательность преобразований состояния модели. Например, вы можете чередовать значение столбца `admin` между `Y` и `N` для каждого вновь созданного пользователя:

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
    ->count(10)
    ->state(new Sequence(
        ['admin' => 'Y'],
        ['admin' => 'N'],
    ))
```

```
    ))  
->create();
```

В этом примере пять пользователей будут созданы со значением `admin`, равным `Y`, и пять пользователей – со значением `admin`, равным `N`.

При необходимости вы можете внедрить замыкание в качестве значения последовательности. Замыкание будет вызываться каждый раз, когда последовательности потребуется новое значение:

```
use Illuminate\Database\Eloquent\Factories\Sequence;  
  
$users = User::factory()  
    ->count(10)  
    ->state(new Sequence(  
        fn (Sequence $sequence) => ['role' => UserRoles::all()->random()])  
    )  
    ->create();
```



Внутри замыкания последовательности вы можете получить доступ к свойствам `$index` или `$count` экземпляра последовательности, который вводится в замыкание. Свойство `$index` содержит номер текущей итерации, а свойство `$count` – общее количество итераций:

```
$users = User::factory()  
    ->count(10)  
    ->sequence(fn (Sequence $sequence) => ['name' => 'Name ' . $sequence->  
    ->create();
```



Для удобства последовательности также можно применять с использованием метода `sequence`, который внутренне просто вызывает метод `state`. Метод `sequence` принимает замыкание или массивы атрибутов для последовательности:

```
$users = User::factory()  
    ->count(2)  
    ->sequence(  
        ['name' => 'First User'],  
        ['name' => 'Second User'],
```

```
)  
->create();
```

## # Отношения

### Отношения Has Many

Теперь давайте рассмотрим построение отношений моделей Eloquent с использованием текущего интерфейса методов фабрик Laravel. Во-первых, предположим, что у нашего приложения есть модель `App\Models\User` и модель `App\Models\Post`. Также предположим, что модель `User` определяет отношения `hasMany` с `Post`. Мы можем создать пользователя с тремя постами, используя метод `has`, предоставляемый фабриками Laravel. Метод `has` принимает экземпляр фабрики:

```
use App\Models\Post;  
use App\Models\User;  
  
$user = User::factory()  
    ->has(Post::factory()->count(3))  
    ->create();
```

По соглашению, при передаче модели `Post` методу `has`, Laravel будет предполагать, что модель `User` должна иметь метод `posts`, который определяет отношения. При необходимости вы можете явно указать имя отношения, которым вы хотите управлять:

```
$user = User::factory()  
    ->has(Post::factory()->count(3), 'posts')  
    ->create();
```

Конечно, вы можете выполнять манипуляции с состоянием связанных моделей. Кроме того, вы можете преобразовать состояние связанной модели с помощью замыкания, предоставив ему доступ к родительской модели:

```
$user = User::factory()  
    ->has(  
        Post::factory()  
            ->count(3)  
            ->state(function (array $attributes, User $user) {
```

```
        return [ 'user_type' => $user->type];
    })
->create();
```

## Использование магических методов Has Many

Для удобства вы можете использовать магические методы отношений фабрики Laravel для построения отношений. Например, в следующем примере будет использоваться соглашение, определяющее, что связанные модели должны быть созданы с помощью метода отношений `posts` модели `User`:

```
$user = User::factory()
    ->hasPosts(3)
    ->create();
```

При использовании магических методов для создания отношений фабрики вы можете передать массив атрибутов для их переопределения в связанных моделях:

```
$user = User::factory()
    ->hasPosts(3, [
        'published' => false,
    ])
    ->create();
```

Вы можете преобразовать состояние связанной модели с помощью замыкания, предоставив ему доступ к родительской модели:

```
$user = User::factory()
    ->hasPosts(3, function (array $attributes, User $user) {
        return [ 'user_type' => $user->type];
    })
    ->create();
```

## Отношения Belongs To

Теперь, когда мы изучили, как построить отношения Has Many с помощью фабрик, давайте рассмотрим обратное отношение. Метод `for` используется для определения родительской модели, к которой принадлежат модели, созданные

фабрикой. Например, мы можем создать три экземпляра модели `App\Models\Post`, которые принадлежат одному пользователю:

```
use App\Models\Post;
use App\Models\User;

$posts = Post::factory()
    ->count(3)
    ->for(User::factory()->state([
        'name' => 'Jessica Archer',
    ]))
    ->create();
```

Если у вас уже есть экземпляр родительской модели, который должен быть связан с создаваемыми вами моделями, вы можете передать экземпляр модели методу `for`:

```
$user = User::factory()->create();

$posts = Post::factory()
    ->count(3)
    ->for($user)
    ->create();
```

## Использование магических методов Belongs To

Для удобства вы можете использовать магические методы отношений фабрики Laravel для построения отношений Belongs To. Например, в следующем примере будет использоваться соглашение, чтобы определить, что три поста должны принадлежать отношениям `user` в модели `Post`:

```
$posts = Post::factory()
    ->count(3)
    ->forUser([
        'name' => 'Jessica Archer',
    ])
    ->create();
```

## Отношения Many To Many

Как и [отношения Has Many](#), отношения Many To Many могут быть созданы с использованием метода `has`:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
    ->has(Role::factory()->count(3))
    ->create();
```

## Атрибуты сводной таблицы

Если вам нужно определить атрибуты, которые должны быть установлены в сводной / промежуточной таблице, связывающей модели, вы можете использовать метод `hasAttached`. Этот метод принимает в качестве второго аргумента массив имен и значений атрибутов сводной таблицы:

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
    ->hasAttached(
        Role::factory()->count(3),
        ['active' => true]
    )
    ->create();
```

Вы можете преобразовать состояние связанной модели с помощью замыкания, предоставив ему доступ к родительской модели:

```
$user = User::factory()
    ->hasAttached(
        Role::factory()
            ->count(3)
            ->state(function (array $attributes, User $user) {
                return ['name' => $user->name. ' Role'];
            }),
        ['active' => true]
    )
    ->create();
```

Если у вас уже есть экземпляры модели, которые вы хотите прикрепить к создаваемым моделям, вы можете передать экземпляры модели методу `hasAttached`. В этом примере всем трем пользователям будут назначены одни и те же три роли:

```
$roles = Role::factory()->count(3)->create();  
  
$user = User::factory()  
    ->count(3)  
    ->hasAttached($roles, ['active' => true])  
    ->create();
```

## Использование магических методов Many To Many

Для удобства вы можете использовать магические методы отношений фабрики Laravel для построения отношений Many To Many. Например, в следующем примере будет использоваться соглашение, чтобы определить, что связанные модели должны быть созданы с помощью метода отношений `roles` модели `User`:

```
$user = User::factory()  
    ->hasRoles(1, [  
        'name' => 'Editor'  
    ])  
    ->create();
```

## Полиморфные отношения

[Полиморфные отношения](#) также могут быть созданы с использованием фабрик. Полиморфные отношения Morph Many создаются так же, как типичные отношения Has Many. Например, если модель `App\Models\Post` имеет отношение `morphMany` с моделью `App\Models\Comment`:

```
use App\Models\Post;  
  
$post = Post::factory()->hasComments(3)->create();
```

## Отношения Morph To

Магические методы нельзя использовать для создания отношений `morphTo`. Вместо этого метод `for` должен использоваться напрямую, а имя отношения должно быть явно указано. Например, представьте, что модель `Comment` имеет метод `commentable`, который определяет отношение `morphTo`. В этой ситуации мы можем создать три комментария, относящиеся к одному посту, используя напрямую метод `for`:

```
$comments = Comment::factory()->count(3)->for(
    Post::factory(), 'commentable'
)->create();
```

## Полиморфные отношения Many To Many

Полиморфные отношения Many To Many (`morphToMany` / `morphedByMany`) могут быть созданы точно так же, как не полиморфные отношения Many To Many:

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
    ->hasAttached(
        Tag::factory()->count(3),
        ['public' => true]
    )
    ->create();
```

Конечно, магический метод `has` также используется для создания полиморфных отношений Many To Many:

```
$videos = Video::factory()
    ->hasTags(3, ['public' => true])
    ->create();
```

## Определение отношений внутри фабрик

Чтобы определить отношение в рамках вашей фабрики модели, вы обычно назначаете новый экземпляр фабрики внешнему ключу отношения. Обычно это делается для «обратных» отношений, таких как `belongsTo` и `morphTo`. Например, если вы хотите создать нового пользователя при создании публикации, вы можете сделать следующее:

```

use App\Models\User;

/**
 * Определить состояние модели по умолчанию.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
        'title' => fake()->title(),
        'content' => fake()->paragraph(),
    ];
}

```

Если свойства отношения зависят от фабрики, которая его определяет, вы можете назначить замыкание атрибуту. Замыкание получит массив проанализированных атрибутов фабрики:

```

/**
 * Определить состояние модели по умолчанию.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
        'user_type' => function (array $attributes) {
            return User::find($attributes['user_id'])->type;
        },
        'title' => fake()->title(),
        'content' => fake()->paragraph(),
    ];
}

```

## Повторное использование существующей модели для отношений

Если у вас есть модели, которые имеют общее отношение с другой моделью, вы можете использовать метод `recycle`, чтобы обеспечить повторное использование одного экземпляра связанной модели для всех отношений, созданных фабрикой.

Например, представьте, что у вас есть модели `Airline`, `Flight`, и `Ticket`, где билет принадлежит авиакомпании и рейсу, а рейс также принадлежит авиакомпании. При создании билетов вы, вероятно, захотите использовать одну и ту же авиакомпанию как для билета, так и для рейса. Для этого вы можете передать экземпляр авиакомпании методу `recycle`:

```
Ticket::factory()  
->recycle(Airline::factory()->create())  
->create();
```

Метод `recycle` может быть особенно полезен, если у вас есть модели, принадлежащие одному пользователю или команде.

Метод `recycle` также принимает коллекцию существующих моделей. Если методу `recycle` предоставляется коллекция, то, когда фабрике понадобится модель данного типа, из коллекции будет выбрана случайная модель:

```
Ticket::factory()  
->recycle($airlines)  
->create();
```

# Тестирование · Начало работы

- # Введение
- # Окружение
- # Создание тестов
- # Запуск тестов

- # Параллельное выполнение тестов
- # Отчет о покрытии тестами
- # Профилирование тестов

## # Введение

Laravel построен с учетом требований тестирования. Поддержка тестирования с помощью PHPUnit включена прямо из коробки, и файл `phpunit.xml` уже настроен для вашего приложения. Фреймворк также поставляется с удобными вспомогательными методами, позволяющими выразительно тестировать ваши приложения.

По умолчанию каталог `tests` вашего приложения содержит два каталога: `Feature` и `Unit`. Модульные (юнит) тесты – это тесты, которые фокусируются на очень небольшой изолированной части вашего кода. Фактически, большинство модульных тестов, вероятно, сосредоточены на одном методе. Тесты в каталоге «Unit» тестов не загружают ваше приложение Laravel и, следовательно, не могут получить доступ к базе данных вашего приложения или другим службам фреймворка.

Функциональные тесты могут тестировать большую часть вашего кода, включая взаимодействие нескольких объектов друг с другом, или даже целый HTTP-запрос, возвращающий JSON. **Как правило, большинство ваших тестов должны быть функциональными. Эти типы тестов обеспечивают максимальную уверенность в том, что ваша система в целом работает должным образом.**

Файл `ExampleTest.php` находится в каталогах тестов `Feature` и `Unit`. После установки нового приложения Laravel выполните команды `vendor/bin/phpunit` или `php artisan`

`test` из командной строки для запуска ваших тестов.

## # Окружение

При запуске тестов Laravel автоматически устанавливает [конфигурацию окружения](#) в `testing` благодаря переменной окружения, определенной в файле `phpunit.xml`. Laravel также автоматически настраивает сеанс и кеш для драйвера `array`, чтобы данные сеанса или кэша не сохранялись во время тестирования.

При необходимости вы можете определять другие значения конфигурации среды тестирования. Переменные окружения `testing` могут быть настроены в файле `phpunit.xml`, но перед запуском тестов не забудьте очистить кеш конфигурации с помощью Artisan-команды `config:clear`.

### Переменная окружения `.env.testing`

Кроме того, вы можете создать файл `.env.testing` в корне вашего проекта. Этот файл будет использоваться вместо `.env` при запуске тестов PHPUnit или выполнении команд Artisan с параметром `--env=testing`.

### Трейт `CreatesApplication`

Laravel содержит трейт `CreatesApplication`, который применяется к базовому классу `TestCase` вашего приложения. Этот трейт содержит метод `createApplication`, который загружает приложение Laravel перед запуском ваших тестов. Важно, чтобы вы оставили этот трейт в его исходном месте, так как от него зависит некоторый функционал, например, функционал параллельного тестирования Laravel.

## # Создание тестов

Чтобы сгенерировать новый тест, используйте [Artisan](#)-команду `make:test`. Эта команда поместит новый класс теста в каталог `tests/Feature` вашего приложения:

```
php artisan make:test UserTest
```

Если вы хотите создать тест в каталоге `tests/Unit`, то используйте параметр `--unit` при выполнении команды `make:test`:

```
php artisan make:test UserTest --unit
```

Если вы хотите создать тест [Pest PHP](#), вы можете указать параметр `--pest` для команды `make:test`:

```
php artisan make:test UserTest --pest
php artisan make:test UserTest --unit --pest
```

Заготовки тестов можно настроить с помощью [публикации заготовок](#).

После того как тест был сгенерирован, вы можете определить методы тестирования, как обычно, используя [PHPUnit](#). Чтобы запустить ваши тесты, выполните команду `vendor/bin/phpunit` или `php artisan test` из вашего терминала:

```
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * Отвлеченный пример модульного теста.
     */
    public function test_basic_test(): void
    {
        $this->assertTrue(true);
    }
}
```

Если вы определяете свои собственные методы `setUp` / `tearDown` в тестовом классе, обязательно вызывайте соответствующие методы `parent::setUp()` / `parent::tearDown()` родительского класса. Обычно

вы должны вызывать `parent::setUp()` в начале своего собственного метода `setUp`, а `parent::tearDown()` в конце вашего метода `tearDown`.

## # Запуск тестов

Как упоминалось ранее, после того, как вы написали тесты, вы можете запускать их с помощью `pest` или `phpunit`:

Pest      PHPUnit

```
./vendor/bin/pest
```

В дополнение к командам `pest` или `phpunit`, вы можете использовать команду `test` Artisan для запуска ваших тестов. Тестер Artisan отображает подробные отчеты о тестах для упрощения разработки и отладки:

```
php artisan test
```

Любые аргументы, которые могут быть переданы командам `pest` или `phpunit`, также могут быть переданы команде Artisan `test`:

```
php artisan test --testsuite=Feature --stop-on-failure
```

## Параллельное выполнение тестов

По умолчанию Laravel и PHPUnit выполняют ваши тесты последовательно в рамках одного процесса. Однако вы можете значительно сократить время выполнения ваших тестов, запуская их параллельно в нескольких процессах. Для начала вам следует установить пакет Composer `brianium/paratest` как "dev" зависимость. Затем при выполнении команды `test` Artisan вы должны включить опцию `--parallel`:

```
composer require brianium/paratest --dev
```

```
php artisan test --parallel
```

По умолчанию Laravel создает столько процессов, сколько ядер ЦП доступно на вашем компьютере. Однако вы можете настроить количество процессов, используя параметр `--processes`:

```
php artisan test --parallel --processes=4
```

При параллельном запуске тестов некоторые параметры PHPUnit (такие, как `--do-not-cache-result`) могут быть недоступны.

## Параллельное тестирование и базы данных

При условии, что вы настроили основное подключение к базе данных, Laravel автоматически обрабатывает создание и миграцию тестовой базы данных для каждого параллельного процесса, в котором выполняются ваши тесты. К тестовым базам данных будет добавлен суффикс, уникальный для каждого процесса. Например, если у вас есть два параллельных тестовых процесса, Laravel создаст и будет использовать тестовые базы данных `your_db_test_1` и `your_db_test_2`.

По умолчанию тестовые базы данных сохраняются между вызовами команды `test` Artisan, чтобы их можно было использовать снова при последующих вызовах `test`. Однако вы можете пересоздать их, используя параметр `--recreate-databases`:

```
php artisan test --parallel --recreate-databases
```

## Хуки параллельного тестирования

Иногда требуется подготовить определенные ресурсы, используемые тестами вашего приложения, чтобы их можно было безопасно использовать в нескольких процессах тестирования.

Используя фасад `ParallelTesting`, вы можете указать код, который будет выполняться в `setUp` и `tearDown` процесса или тестового класса. Переданные замыкания получат переменные `$token` и `$testCase`, которые содержат токен процесса и текущий тестовый класс, соответственно:

```
<?php
```

```
namespace App\Providers;

use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\ParallelTesting;
use Illuminate\Support\ServiceProvider;
use PHPUnit\Framework\TestCase;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Загрузка любых служб приложения.
     */
    public function boot(): void
    {
        ParallelTesting::setUpProcess(function (int $token) {
            // ...
        });

        ParallelTesting::setUpTestCase(function (int $token, TestCase $testCase) {
            // ...
        });

        // Выполнится при создании тестовой базы данных ...
        ParallelTesting::setUpTestDatabase(function (string $database, int $token) {
            Artisan::call('db:seed');
        });

        ParallelTesting::tearDownTestCase(function (int $token, TestCase $testCase) {
            // ...
        });

        ParallelTesting::tearDownProcess(function (int $token) {
            // ...
        });
    }
}
```

## Доступ к токену процесса параллельного тестирования

Если вы хотите получить доступ к текущему “токену” параллельного процесса из любого другого места в коде вашего приложения для тестирования, вы можете использовать метод `token`. Этот токен представляет собой уникальный строковый идентификатор для каждого индивидуального тестового процесса и может

использоваться для сегментации ресурсов между параллельными тестовыми процессами. Например, Laravel автоматически добавляет этот токен в конец имен тестовых баз данных, созданных каждым параллельным тестовым процессом:

```
$token = ParallelTesting::token();
```

## Отчет о покрытии тестами

Для использования этой функции требуется [Xdebug](#) или [PCOV](#).

При запуске тестов вашего приложения вам может потребоваться определить, действительно ли ваши тесты охватывают код приложения и насколько много кода приложения используется при выполнении ваших тестов. Для этого вы можете использовать опцию `--coverage` при вызове команды `test`:

```
php artisan test --coverage
```

## Установка минимального порога покрытия

Вы можете использовать опцию `--min`, чтобы задать минимальный порог покрытия тестами для вашего приложения. Процесс завершит выполнение с ошибкой, если этот порог не будет достигнут:

```
php artisan test --coverage --min=80.3
```

## Профилирование тестов

Запуская тестовый раннер Artisan, вы также можете узнать, какие из ваших тестов работают медленно. Вызовите команду `test` с опцией `--profile`, чтобы получить список десяти самых медленных тестов вашего приложения, что позволит вам легко выяснить, какие тесты можно улучшить, чтобы ускорить выполнение:

```
php artisan test --profile
```

# Тестирование · Тесты HTTP

## # Введение

## # Выполнение запросов

- # Настройка заголовков запросов
- # Cookies
- # Сессия / Аутентификация
- # Отладка ответов
- # Обработка исключений

## # Тестирование JSON API

- # Последовательное тестирование JSON

## # Тестирование загрузки файлов

## # Тестирование шаблонной системы

- # Отрисовка Blade и компоненты

## # Доступные утверждения

- # Утверждения ответов
- # Утверждения аутентификации

## # Утверждения валидации

## # Введение

Laravel предлагает гибкий API в составе вашего приложения для выполнения HTTP-запросов и получения информации об ответах. Например, взгляните на следующий функциональный тест:

Pest      PHPUnit

```
<?php
```

```
test('the application returns a successful response', function () {  
    $response = $this->get('/');
```

```
$response->assertStatus(200);  
});
```

Метод `get` отправляет в приложение запрос `GET`, а метод `assertStatus` утверждает, что возвращаемый ответ должен иметь указанный код состояния HTTP. Помимо этого простого утверждения, Laravel также содержит множество утверждений для получения информации о заголовках ответов, их содержимого, структуры JSON и др.

## # Выполнение запросов

Чтобы сделать запрос к вашему приложению, вы можете вызвать в своем тесте методы `get`, `post`, `put`, `patch`, или `delete`. Эти методы фактически не отправляют вашему приложению «настоящий» HTTP-запрос. Вместо этого внутри моделируется полный сетевой запрос.

Вместо того чтобы возвращать экземпляр `Illuminate\Http\Response`, методы тестового запроса возвращают экземпляр `Illuminate\Testing\TestResponse`, который содержит множество полезных утверждений, позволяющие вам инспектировать ответы вашего приложения:

Pest      PHPUnit

```
<?php  
  
test('basic request', function () {  
    $response = $this->get('/');  
  
    $response->assertStatus(200);  
});
```

Как правило, каждый из ваших тестов должен выполнять только один запрос к вашему приложению. Неожиданное поведение может возникнуть, если в рамках одного метода теста выполняется несколько запросов.

Для удобства посредник CSRF автоматически отключается при запуске тестов.

## Настройка заголовков запросов

Вы можете использовать метод `withHeaders` для настройки заголовков запроса перед его отправкой в приложение. Этот метод позволяет вам добавлять в запрос любые пользовательские заголовки:

Pest      PHPUnit

```
<?php

test('interacting with headers', function () {
    $response = $this->withHeaders([
        'X-Header' => 'Value',
    ])->post('/user', ['name' => 'Sally']);

    $response->assertStatus(201);
});
```

## Cookies

Вы можете использовать методы `withCookie` или `withCookies` для установки значений файлов Cookies перед отправкой запроса. Метод `withCookie` принимает имя и значение Cookie в качестве двух аргументов, а метод `withCookies` принимает массив пар имя / значение:

Pest      PHPUnit

```
<?php

test('interacting with cookies', function () {
    $response = $this->withCookie('color', 'blue')->get('/');

    $response = $this->withCookies([
        'color' => 'blue',
        'name' => 'Taylor',
    ])->get('/');

    //
});
```

## Сессия / Аутентификация

Laravel предлагает несколько методов-хелперов для взаимодействия с сессией во время HTTP-тестирования. Во-первых, вы можете установить данные сессии, передав массив, используя метод `withSession`. Это полезно для загрузки сессии данными перед отправкой запроса вашему приложению:

Pest      PHPUnit

```
<?php

test('interacting with the session', function () {
    $response = $this->withSession(['banned' => false])->get('/');
    // ...
});
```

Сессия Laravel обычно используется для сохранения состояния текущего аутентифицированного пользователя. Вспомогательный метод `actingAs` – это простой способ аутентифицировать конкретного пользователя как текущего. Например, мы можем использовать [фабрику модели](#) для генерации и аутентификации пользователя:

Pest      PHPUnit

```
<?php

use App\Models\User;

test('an action that requires authentication', function () {
    $user = User::factory()->create();

    $response = $this->actingAs($user)
        ->withSession(['banned' => false])
        ->get('/');
    // ...
});
```

Вы также можете указать, какой гейт должен использоваться для аутентификации конкретного пользователя, передав имя гейта в качестве второго аргумента методу `actingAs`. Гейт, предоставленный методу `actingAs`, также станет гейтом по умолчанию на протяжении всего теста::

```
$this->actingAs($user, 'web')
```

## Отладка ответов

После выполнения тестового запроса к вашему приложению методы `dump`, `dumpHeaders`, и `dumpSession` могут быть использованы для проверки и отладки содержимого ответа:

Pest      PHPUnit

```
<?php

test('basic test', function () {
    $response = $this->get('/');

    $response->dumpHeaders();

    $response->dumpSession();

    $response->dump();
});
```

В качестве альтернативы вы можете использовать методы `dd`, `ddHeaders` и `ddSession`, чтобы выгрузить информацию об ответе и затем остановить выполнение:

Pest      PHPUnit

```
<?php

test('basic test', function () {
    $response = $this->get('/');

    $response->ddHeaders();

    $response->ddSession();

    $response->dd();
});
```

## Обработка исключений

Иногда вам может понадобиться проверить, выдает ли ваше приложение определенное исключение. Для этого вы можете «подделать» обработчик исключений через фасад `Exceptions`. После того как обработчик исключений был подделан, вы можете использовать методы `assertReported` и `assertNotReported` для создания утверждений против исключений, которые были созданы во время запроса:

Pest      PHPUnit

<?php

```
use App\Exceptions\InvalidOrderException;
use Illuminate\Support\Facades\Exceptions;

test('exception is thrown', function () {
    Exceptions::fake();

    $response = $this->get('/order/1');

    // Assert an exception was thrown...
    Exceptions::assertReported(InvalidOrderException::class);

    // Assert against the exception...
    Exceptions::assertReported(function (InvalidOrderException $e) {
        return $e->getMessage() === 'The order was invalid.';
    });
});
```

Методы `assertNotReported` и `assertNothingReported` могут использоваться для подтверждения того, что данное исключение не было создано во время запроса или что никаких исключений не было создано:

```
Exceptions::assertNotReported(InvalidOrderException::class);

Exceptions::assertNothingReported();
```

Вы можете полностью отключить обработку исключений для данного запроса, вызвав метод `withoutExceptionHandling` перед отправкой запроса:

```
$response = $this->withoutExceptionHandling()->get('/');
```

Кроме того, если вы хотите убедиться, что ваше приложение не использует функции, считающиеся устаревшими языком PHP или библиотеками, которые использует ваше приложение, вы можете вызвать метод `withoutDeprecationHandling` перед тем, как сделать свой запрос. Когда обработка устаревания отключена, предупреждения об устаревании будут преобразованы в исключения, что приведет к сбою вашего теста:

```
$response = $this->withoutDeprecationHandling()->get('/');
```

Метод `assertThrows` можно использовать для проверки того, что код внутри заданного замыкания генерирует исключение [указанного](#) типа:

```
$this->assertThrows(
    fn () => (new ProcessOrder)->execute(),
    OrderInvalid::class
);
```

Если вы хотите проверить и сделать утверждения против выброшенного исключения, вы можете предоставить замыкание в качестве второго аргумента метода `assertThrows`:

```
$this->assertThrows(
    fn () => (new ProcessOrder)->execute(),
    fn (OrderInvalid $e) => $e->orderId() === 123;
);
```

## # Тестирование JSON API

Laravel также содержит несколько хелперов для тестирования API-интерфейсов JSON и их ответов. Например, методы `json`, `getJson`, `postJson`, `putJson`, `patchJson`, `deleteJson`, и `optionsJson` могут использоваться для отправки запросов JSON с различными HTTP-командами. Вы также можете передавать данные и заголовки этим методам. Для начала давайте напишем тест, чтобы сделать запрос `POST` к `/api/user` и убедиться, что в JSON были возвращены ожидаемые данные:

```
<?php

test('making an api request', function () {
    $response = $this->postJson('/api/user', ['name' => 'Sally']);

    $response
        ->assertStatus(201)
        ->assertJson([
            'created' => true,
        ]);
});
```

Кроме того, к данным ответа JSON можно получить доступ как к переменным массива в ответе, что позволяет удобно проверять отдельные значения, возвращаемые в JSON-ответе:

Pest      PHPUnit

```
expect($response['created'])->toBeTrue();
```

Метод `assertJson` преобразует ответ в массив для проверки того, что переданный массив существует в ответе JSON, возвращаемом приложением. Итак, если в ответе JSON есть другие свойства, этот тест все равно будет проходить, пока присутствует переданный фрагмент.

## Утверждение точных совпадений JSON

Как упоминалось ранее, метод `assertJson` используется для подтверждения наличия фрагмента JSON в ответе JSON. Если вы хотите убедиться, что данный массив **в точности соответствует** JSON, возвращаемому вашим приложением, вы должны использовать метод `assertExactJson`:

Pest      PHPUnit

```
<?php
```

```
test('asserting an exact json match', function () {
    $response = $this->postJson('/user', ['name' => 'Sally']);

    $response
        ->assertStatus(201)
        ->assertExactJson([
            'created' => true,
        ]);
});
```

## Утверждения в JSON-путях

Если вы хотите убедиться, что ответ JSON содержит данные по указанному пути, вам следует использовать метод `assertJsonPath`:

Pest      PHPUnit

```
<?php

test('asserting a json path value', function () {
    $response = $this->postJson('/user', ['name' => 'Sally']);

    $response
        ->assertStatus(201)
        ->assertJsonPath('team.owner.name', 'Darian');
});
```

Метод `assertJsonPath` также принимает замыкание, которое может быть использовано для динамического определения, должно ли утверждение выполниться:

```
$response->assertJsonPath('team.owner.name', fn (string $name) => strlen($name) >= 3)
```

## Последовательное тестирование JSON

Laravel предлагает способ последовательного тестирования ответов JSON вашего приложения. Для начала передайте замыкание методу `assertJson`. Это замыкание будет вызываться с экземпляром класса `Illuminate\Testing\Fluent\AssertableJson`, который можно использовать для создания утверждений в отношении JSON, возвращенного вашим приложением. Метод `where` может использоваться для

утверждения определенного атрибута JSON, в то время как метод `missing` может использоваться для утверждения отсутствия конкретного атрибута в JSON:

Pest      PHPUnit

```
use Illuminate\Testing\Fluent\AssertableJson;

test('fluent json', function () {
    $response = $this->getJson('/users/1');

    $response
        ->assertJson(fn (AssertableJson $json) =>
            $json->where('id', 1)
                ->where('name', 'Victoria Faith')
                ->where('email', fn (string $email) => str($email)->is('victoria@gm...
                ->whereNot('status', 'pending')
                ->missing('password')
                ->etc()
        );
});
```

## Понимание метода `etc`

В приведенном выше примере вы могли заметить, что мы вызвали метод `etc` в конце нашей цепочки утверждений. Этот метод сообщает Laravel, что в объекте JSON могут присутствовать другие атрибуты. Если метод `etc` не используется, то тест завершится неудачно, если в объекте JSON существуют другие атрибуты, для которых вы не сделали утверждений.

Цель такого поведения – защитить вас от непреднамеренного раскрытия конфиденциальной информации в ваших ответах JSON, заставив вас либо явно сделать утверждение относительно атрибута, либо явно разрешить дополнительные атрибуты с помощью метода `etc`.

Однако вы должны знать, что отсутствие метода `etc` в вашей цепочке утверждений не гарантирует, что дополнительные атрибуты не будут добавлены в массивы, вложенные в ваш объект JSON. Метод `etc` обеспечивает только отсутствие дополнительных атрибутов на уровне вложенности, на котором вызывается метод `etc`.

## Утверждение наличия / отсутствия атрибута

Чтобы утверждать, что атрибут присутствует или отсутствует, вы можете использовать методы `has` и `missing`:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('data')
        ->missing('message')
);
```

Кроме того, методы `hasAll` и `missingAll` позволяют одновременно утверждать наличие или отсутствие нескольких атрибутов:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->hasAll(['status', 'data'])
        ->missingAll(['message', 'code'])
);
```

Вы можете использовать метод `hasAny`, чтобы определить, присутствует ли хотя бы один из заданного списка атрибутов:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('status')
        ->hasAny('data', 'message', 'code')
);
```

## Утверждения относительно коллекций JSON

Часто ваш маршрут возвращает ответ JSON, содержащий несколько элементов, например нескольких пользователей:

```
Route::get('/users', function () {
    return User::all();
});
```

В этих ситуациях можно использовать метод `has` последовательного тестирования JSON, чтобы сделать утверждения относительно пользователей, содержащихся в ответе. Например, предположим, что ответ JSON содержит трех пользователей. Затем мы сделаем некоторые утверждения относительно первого пользователя в коллекции, используя метод `first`. Метод `first` принимает замыкание, получающее

другой экземпляр `AssertibleJson`, который можно использовать для создания утверждений относительно первого объекта коллекции JSON:

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has(3)
            ->first(fn (AssertableJson $json) =>
                $json->where('id', 1)
                    ->where('name', 'Victoria Faith')
                    ->where('email', fn (string $email) => str($email)->is('victoria.f@lucifer.com'))
                    ->missing('password')
                    ->etc()
            )
    );

```

## Уровень вложенности утверждения

### относительно коллекций JSON

Иногда маршрутами вашего приложения могут быть возвращены коллекции JSON, которым назначены именованные ключи:

```
Route::get('/users', function () {
    return [
        'meta' => [...],
        'users' => User::all(),
    ];
})
```

При тестировании этих маршрутов вы можете использовать метод `has` для утверждения относительно количества элементов в коллекции. Кроме того, вы можете использовать метод `has` для определения цепочки утверждений:

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has('meta')
            ->has('users', 3)
            ->has('users.0', fn (AssertableJson $json) =>
                $json->where('id', 1)
                    ->where('name', 'Victoria Faith')
                    ->missing('password')
                    ->etc()
            )
    );

```

```
)  
);
```

Однако вместо того, чтобы делать два отдельных вызова метода `has` для утверждения в отношении коллекции `users`, вы можете сделать один вызов, обеспеченный замыканием в качестве третьего параметра. При этом автоматически вызывается замыкание, область действия которого будет ограничено уровнем вложенности первого элемента коллекции:

```
$response  
    ->assertJson(fn (AssertableJson $json) =>  
        $json->has('meta')  
            ->has('users', 3, fn (AssertableJson $json) =>  
                $json->where('id', 1)  
                    ->where('name', 'Victoria Faith')  
                    ->where('email', fn (string $email) => str($email)->is('victoria'))  
                    ->missing('password')  
                    ->etc()  
            )  
    );
```

## Утверждения относительно типов JSON

При необходимости можно утверждать, что свойства в ответе JSON имеют определенный тип. Класс `Illuminate\Testing\Fluent\AssertableJson` содержит методы `whereType` и `whereAllType`, обеспечивающие простоту таких утверждений:

```
$response->assertJson(fn (AssertableJson $json) =>  
    $json->whereType('id', 'integer')  
        ->whereAllType([  
            'users.0.name' => 'string',  
            'meta' => 'array'  
        ])  
);
```

Можно указать несколько типов в качестве второго параметра метода `whereType`, разделив их символом `|`, или передав массив необходимых типов. Утверждение будет успешно, если значение ответа будет иметь какой-либо из перечисленных типов:

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->whereType('name', 'string|null')
        ->whereType('id', ['string', 'integer'])
);
```

Методы `whereType` и `whereAllType` применимы к следующим типам: `string`, `integer`, `double`, `boolean`, `array`, и `null`.

## # Тестирование загрузки файлов

Класс `Illuminate\Http\UploadedFile` содержит метод `fake`, который можно использовать для создания фиктивных файлов или изображений для тестирования. Это, в сочетании с методом `fake` фасада `Storage`, значительно упрощает тестирование загрузки файлов. Например, вы можете объединить эти две функции, чтобы легко протестировать форму загрузки аватара:

Pest      PHPUnit

```
<?php

use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;

test('avatars can be uploaded', function () {
    Storage::fake('avatars');

    $file = UploadedFile::fake()->image('avatar.jpg');

    $response = $this->post('/avatar', [
        'avatar' => $file,
    ]);

    Storage::disk('avatars')->assertExists($file->hashName());
});
```

Если вы хотите подтвердить, что переданный файл не существует, вы можете использовать метод `assertMissing` фасада `Storage`:

```
Storage::fake('avatars');

// ...
```

```
Storage::disk('avatars')->assertMissing('missing.jpg');
```

## Настройка фиктивного файла

При создании файлов с использованием метода `fake`, предоставляемого классом `UploadedFile`, вы можете указать ширину, высоту и размер изображения (в килобайтах), чтобы лучше протестировать правила валидации вашего приложения:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

Помимо создания изображений, вы можете создавать файлы любого другого типа, используя метод `create`:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

При необходимости вы можете передать аргумент `$mimeType` методу, чтобы явно определить MIME-тип, который должен возвращать файл:

```
UploadedFile::fake()->create(  
    'document.pdf', $sizeInKilobytes, 'application/pdf'  
)
```

## # Тестирование шаблонной системы

Laravel также позволяет отображать шаблоны без имитации HTTP-запроса к приложению. Для этого вы можете вызвать в своем teste метод `view`. Метод `view` принимает имя шаблона и необязательный массив данных. Метод возвращает экземпляр `Illuminate\Testing\TestView`, который предлагает несколько методов для удобных утверждений о содержимом шаблона:

Pest      PHPUnit

```
<?php  
  
test('a welcome view can be rendered', function () {  
    $view = $this->view('welcome', ['name' => 'Taylor']);
```

```
$view->assertSee('Taylor');
});
```

Класс `TestView` содержит следующие методы утверждения: `assertSee`, `assertSeeInOrder`, `assertSeeText`, `assertSeeTextInOrder`, `assertDontSee` и `assertDontSeeText`.

При необходимости вы можете получить необработанное отрисованное содержимое шаблона, преобразовав экземпляр `TestView` в строку:

```
$contents = (string) $this->view('welcome');
```

## Передача ошибок валидации в шаблоны

Некоторые шаблоны могут зависеть от ошибок, хранящихся в [глобальной коллекции ошибок Laravel](#). Чтобы добавить в эту коллекцию сообщения об ошибках, вы можете использовать метод `withErrors`:

```
$view = $this->withErrors([
    'name' => ['Please provide a valid name.']
])->view('form');

$view->assertSee('Please provide a valid name.');
```

## Отрисовка Blade и компоненты

Если необходимо, вы можете использовать метод `blade` для анализа и отрисовки необработанной строки [Blade](#). Подобно методу `view`, метод `blade` возвращает экземпляр [Illuminate\Testing\TestView](#):

```
$view = $this->blade(
    '<x-component :name="$name" />',
    ['name' => 'Taylor']
);

$view->assertSee('Taylor');
```

Вы можете использовать метод `component` для анализа и отрисовки [компоненты Blade](#). Метод `component` возвращает экземпляр [Illuminate\Testing\Component](#):

```
$view = $this->component(Profile::class, ['name' => 'Taylor']);  
  
$view->assertSee('Taylor');
```

## # Доступные утверждения

### Утверждения ответов

Класс `Illuminate\Testing\TestResponse` содержит множество своих методов утверждения, которые вы можете использовать при тестировании вашего приложения. К этим утверждениям можно получить доступ в ответе, возвращаемом тестовыми методами `json`, `get`, `post`, `put`, и `delete`:

[assertAccepted](#)  
[assertBadRequest](#)  
[assertConflict](#)  
[assertCookie](#)  
[assertCookieExpired](#)  
[assertCookieNotExpired](#)  
[assertCookieMissing](#)  
[assertCreated](#)  
[assertDontSee](#)  
[assertDontSeeText](#)  
[assertDownload](#)  
[assertExactJson](#)  
[assertExactJsonStructure](#)  
[assertForbidden](#)  
[assertFound](#)  
[assertGone](#)  
[assertHeader](#)  
[assertHeaderMissing](#)  
[assertInternalServerError](#)  
[assertJson](#)  
[assertJsonCount](#)  
[assertJsonFragment](#)  
[assertJsonIsArray](#)

[assertJsonIsObject](#)  
[assertJsonMissing](#)  
[assertJsonMissingExact](#)  
[assertJsonMissingValidationErrors](#)  
[assertJsonPath](#)  
[assertJsonMissingPath](#)  
[assertJsonStructure](#)  
[assertJsonValidationErrors](#)  
[assertJsonValidationErrorFor](#)  
[assertLocation](#)  
[assertMethodNotAllowed](#)  
[assertMovedPermanently](#)  
[assertContent](#)  
[assertNoContent](#)  
[assertStreamedContent](#)  
[assertNotFound](#)  
[assertOk](#)  
[assertPaymentRequired](#)  
[assertPlainCookie](#)  
[assertRedirect](#)  
[assertRedirectContains](#)  
[assertRedirectToRoute](#)  
[assertRedirectToSignedRoute](#)  
[assertRequestTimeout](#)  
[assertSee](#)  
[assertSeeInOrder](#)  
[assertSeeText](#)  
[assertSeeTextInOrder](#)  
[assertServerError](#)  
[assertServiceUnavailable](#)  
[assertSessionHas](#)  
[assertSessionHasInput](#)  
[assertSessionHasAll](#)  
[assertSessionHasErrors](#)  
[assertSessionHasErrorsIn](#)  
[assertSessionHasNoErrors](#)

[assertSessionDoesntHaveErrors](#)  
[assertSessionMissing](#)  
[assertStatus](#)  
[assertSuccessful](#)  
[assertTooManyRequests](#)  
[assertUnauthorized](#)  
[assertUnprocessable](#)  
[assertUnsupportedMediaType](#)  
[assertValid](#)  
[assertInvalid](#)  
[assertViewHas](#)  
[assertViewHasAll](#)  
[assertViewIs](#)  
[assertViewMissing](#)

## assertBadRequest

Утверждает, что ответ имеет код **400** состояния HTTP – **bad request**:

```
$response->assertBadRequest();
```

## assertAccepted

Утверждает, что ответ имеет код **202** состояния HTTP – **accepted**:

```
$response->assertAccepted();
```

## assertConflict

Утверждает, что ответ имеет код **409** состояния HTTP – **conflict**:

```
$response->assertConflict();
```

## assertCookie

Утверждает, что ответ содержит переданный cookie:

```
$response->assertCookie($cookieName, $value = null);
```

## assertCookieExpired

Утверждает, что в ответе содержится переданный cookie и срок его действия истек:

```
$response->assertCookieExpired($cookieName);
```

## assertCookieNotExpired

Утверждает, что в ответе содержится переданный cookie и срок его действия не истек:

```
$response->assertCookieNotExpired($cookieName);
```

## assertCookieMissing

Утверждает, что ответ не содержит переданный cookie:

```
$response->assertCookieMissing($cookieName);
```

## assertCreated

Утверждает, что ответ имеет код **201** состояния HTTP:

```
$response->assertCreated();
```

## assertDontSee

Утверждает, что переданная строка не содержится в ответе, возвращаемом приложением. Это утверждение автоматически экранирует переданную строку, если вы не передадите второй аргумент как **false**:

```
$response->assertDontSee($value, $escaped = true);
```

## assertDontSeeText

Утверждает, что переданная строка не содержится в тексте ответа. Это утверждение автоматически экранирует переданную строку, если вы не передадите второй аргумент как `false`. Этот метод передаст содержимое ответа PHP-функции `strip_tags` перед тем, как выполнить утверждение:

```
$response->assertDontSeeText($value, $escaped = true);
```

## assertDownload

Утверждение, что ответ является отдачей файла. Обычно это означает, что вызванный маршрут, который вернул ответ, вернул ответ `Response::download`, `BinaryFileResponse` или `Storage::download`:

```
$response->assertDownload();
```

При желании вы можете сделать утверждение, что загружаемому файлу было присвоено данное имя файла:

```
$response->assertDownload('image.jpg');
```

## assertExactJson

Утверждает, что ответ содержит точное совпадение указанных данных JSON:

```
$response->assertExactJson(array $data);
```

## assertExactJsonStructure

Убедитесь, что ответ содержит точное соответствие заданной структуре JSON:

```
$response->assertExactJsonStructure(array $data);
```

Этот метод является более строгим вариантом `assertJsonStructure`. В отличие от `assertJsonStructure`, этот метод завершится ошибкой, если ответ содержит какие-

либо ключи, которые явно не включены в ожидаемую структуру JSON.

## assertForbidden

Утверждает, что ответ имеет код **403** состояния HTTP – **forbidden**:

```
$response->assertForbidden();
```

## assertFound

Утверждает, что ответ имеет код **302** состояния HTTP – **found**:

```
$response->assertFound();
```

## assertGone

Утверждает, что ответ имеет код **420** состояния HTTP – **gone**:

```
$response->assertGone();
```

## assertHeader

Утверждает, что переданный заголовок и значение присутствуют в ответе:

```
$response->assertHeader($headerName, $value = null);
```

## assertHeaderMissing

Утверждает, что переданный заголовок отсутствует в ответе:

```
$response->assertHeaderMissing($headerName);
```

## assertInternalServerError

Утверждает, что ответ имеет код **500** состояния HTTP – **Internal Server Error**:

```
$response->assertInternalServerError();
```

## assertJson

Утверждает, что ответ содержит указанные данные JSON:

```
$response->assertJson(array $data, $strict = false);
```

Метод `assertJson` преобразует ответ в массив для проверки того, что переданный массив существует в ответе JSON, возвращаемом приложением. Итак, если в ответе JSON есть другие свойства, этот тест все равно будет проходить, пока присутствует переданный фрагмент.

## assertJsonCount

Утверждает, что ответ JSON имеет массив с ожидаемым количеством элементов указанного ключа:

```
$response->assertJsonCount($count, $key = null);
```

## assertJsonFragment

Утверждает, что ответ содержит указанные данные JSON в любом месте ответа:

```
Route::get('/users', function () {
    return [
        'users' => [
            [
                'name' => 'Taylor Otwell',
            ],
        ],
    ];
});

$response->assertJsonFragment(['name' => 'Taylor Otwell']);
```

## assertJsonIsArray

Утверждает, что ответ JSON представляет собой массив:

```
$response->assertJsonIsArray();
```

## assertJsonIsObject

Утверждает, что ответ JSON представляет собой объект:

```
$response->assertJsonIsObject();
```

## assertJsonMissing

Утверждает, что ответ не содержит указанных данных JSON:

```
$response->assertJsonMissing(array $data);
```

## assertJsonMissingExact

Утверждает, что ответ не содержит точных указанных данных JSON:

```
$response->assertJsonMissingExact(array $data);
```

## assertJsonMissingValidationErrors

Утверждает, что ответ не содержит ошибок валидации JSON для переданных ключей:

```
$response->assertJsonMissingValidationErrors($keys);
```

Более общий метод [assertValid](#) может использоваться для подтверждения того, что в ответе нет ошибок проверки, которые были возвращены как JSON и что ошибки не были записаны в хранилище сеанса.

## assertJsonPath

Утверждает, что ответ содержит конкретные данные по указанному пути:

```
$response->assertJsonPath($path, $expectedValue);
```

Например, если ваше приложение возвращает следующий ответ JSON:

```
{  
    "user": {  
        "name": "Steve Schoger"  
    }  
}
```

Вы можете утверждать, что свойство `name` объекта `user` соответствует переданному значению следующим образом:

```
$response->assertJsonPath('user.name', 'Steve Schoger');
```

## assertJsonMissingPath

Утверждает, что ответ не содержит указанного пути:

```
$response->assertJsonMissingPath($path);
```

Например, если ваше приложение возвращает следующий ответ JSON:

```
{  
    "user": {  
        "name": "Steve Schoger"  
    }  
}
```

Вы можете утверждать, что ответ не содержит свойства `email` объекта `user`:

```
$response->assertJsonMissingPath('user.email');
```

## assertJsonStructure

Утверждает, что ответ имеет переданную структуру JSON:

```
$response->assertJsonStructure(array $structure);
```

Например, если ответ JSON, возвращаемый вашим приложением, содержит следующие данные:

```
{  
    "user": {  
        "name": "Steve Schoger"  
    }  
}
```

Вы можете утверждать, что структура JSON соответствует вашим ожиданиям, например:

```
$response->assertJsonStructure([  
    'user' => [  
        'name',  
    ]  
]);
```

Иногда ответы JSON, возвращаемые вашим приложением, могут содержать массивы объектов:

```
{  
    "user": [  
        {  
            "name": "Steve Schoger",  
            "age": 55,  
            "location": "Earth"  
        },  
        {  
            "name": "Mary Schoger",  
            "age": 60,  
            "location": "Earth"  
        }  
    ]
```

```
]  
}
```

В этой ситуации вы можете использовать символ `*` для утверждения о структуре всех объектов в массиве:

```
$response->assertJsonStructure([  
    'user' => [  
        '*' => [  
            'name',  
            'age',  
            'location'  
        ]  
    ]  
]);
```

## assertJsonValidationErrors

Утверждает, что ответ содержит переданные ошибки валидации JSON для переданных ключей. Этот метод следует использовать при утверждении ответов, в которых ошибки валидации возвращаются как структура JSON, а не кратковременно передаются в сессию:

```
$response->assertJsonValidationErrors(array $data, $responseKey = 'errors');
```

Более общий метод [assertInvalid](#) может использоваться для подтверждения того, что в ответе есть ошибки проверки, возвращенные как JSON **или** что ошибки были записаны в хранилище сеанса.

## assertJsonValidationErrorFor

Утверждает, что в ответе есть какие-либо ошибки проверки JSON для данного ключа:

```
$response->assertJsonValidationErrorsFor(string $key, $responseKey = 'errors');
```

## assertMethodNotAllowed

Утверждает, что ответ имеет код **405** состояния HTTP – **method not allowed**:

```
$response->assertMethodNotAllowed();
```

## assertMovedPermanently

Утверждает, что ответ имеет код **301** состояния HTTP – **moved permanently**:

```
$response->assertMovedPermanently();
```

## assertLocation

Утверждает, что ответ имеет переданное значение URI в заголовке **Location**:

```
$response->assertLocation($uri);
```

## assertContent

Утверждает, что указанная строка соответствует содержимому ответа:

```
$response->assertContent($value);
```

## assertNoContent

Утверждает, что ответ имеет код **204** состояния HTTP – **no content**:

```
$response->assertNoContent($status = 204);
```

## assertStreamedContent

Утверждает, что указанная строка соответствует потоковому содержимому ответа:

```
$response->assertStreamedContent($value);
```

## assertNotFound

Утверждает, что ответ имеет код **404** состояния HTTP – **not found**:

```
$response->assertNotFound();
```

## assertOk

Утверждает, что ответ имеет код **200** состояния HTTP – **OK**:

```
$response->assertOk();
```

## assertPaymentRequired

Утверждает, что ответ имеет код **402** состояния HTTP – **payment required**:

```
$response->assertPaymentRequired();
```

## assertPlainCookie

Утверждает, что ответ содержит переданный незашифрованный cookie:

```
$response->assertPlainCookie($cookieName, $value = null);
```

## assertRedirect

Утверждает, что ответ является перенаправлением на указанный URI:

```
$response->assertRedirect($uri = null);
```

## assertRedirectContains

Утверждает, перенаправляет ли ответ на URI, который содержит данную строку:

```
$response->assertRedirectContains($string);
```

## assertRedirectToRoute

Утверждите, что ответ представляет собой перенаправление на указанный именованный маршрут:

```
$response->assertRedirectToRoute($name, $parameters = []);
```

## assertRedirectToSignedRoute

Утверждите, что ответ представляет собой перенаправление на указанный подписанный маршрут:

```
$response->assertRedirectToSignedRoute($name = null, $parameters = []);
```

## assertRequestTimeout

Утверждает, что ответ имеет код **408** состояния HTTP – **request timeout**:

```
$response->assertRequestTimeout();
```

## assertSee

Утверждает, что переданная строка содержится в ответе. Это утверждение автоматически экранирует переданную строку, если вы не передадите второй аргумент как **false**:

```
$response->assertSee($value, $escaped = true);
```

## assertSeeInOrder

Утверждает, что переданные строки содержатся в ответе в указанном порядке. Это утверждение автоматически экранирует переданные строки, если вы не передадите второй аргумент как **false**:

```
$response->assertSeeInOrder(array $values, $escaped = true);
```

## assertSeeText

Утверждает, что переданная строка содержится в тексте ответа. Это утверждение автоматически экранирует переданную строку, если вы не передадите второй аргумент как `false`. Этот метод передаст содержимое ответа PHP-функции `strip_tags` перед тем, как выполнить утверждение:

```
$response->assertSeeText($value, $escaped = true);
```

## assertSeeTextInOrder

Утверждает, что переданные строки содержатся в тексте ответа в указанном порядке. Это утверждение автоматически экранирует переданные строки, если вы не передадите второй аргумент как `false`. Этот метод передаст содержимое ответа PHP-функции `strip_tags` перед тем, как выполнить утверждение:

```
$response->assertSeeTextInOrder(array $values, $escaped = true);
```

## assertServerError

Утверждает, что ответ имеет код состояния HTTP соответствующий ошибке сервера – `>= 500 , < 600`:

```
$response->assertServerError();
```

## assertServiceUnavailable

Утверждает, что ответ имеет код `503` состояния HTTP – `Service Unavailable`:

```
$response->assertServiceUnavailable();
```

## assertSessionHas

Утверждает, что сессия содержит переданный фрагмент данных:

```
$response->assertSessionHas($key, $value = null);
```

Если необходимо, замыкание может быть предоставлено в качестве второго аргумента метода `assertSessionHas`. Утверждение пройдет, если замыкание вернет `true`:

```
 $$response->assertSessionHas($key, function (User $value) {
    return $value->name === 'Taylor Otwell';
});
```

## assertSessionHasInput

Утверждает, что сессия имеет переданное значение в массиве входящих данных кратковременного сохранения:

```
$response->assertSessionHasInput($key, $value = null);
```

Если необходимо, замыкание может быть предоставлено в качестве второго аргумента метода `assertSessionHasInput`. Утверждение пройдет, если замыкание вернет `true`:

```
use Illuminate\Support\Facades\Crypt;
```

```
$response->assertSessionHasInput($key, function (string $value) {
    return Crypt::decryptString($value) === 'secret';
});
```

## assertSessionHasAll

Утверждает, что сессия содержит переданный массив пар ключ / значение:

```
$response->assertSessionHasAll(array $data);
```

Например, если сессия вашего приложения содержит ключи `name` и `status`, вы можете утверждать, что оба они существуют и имеют указанные значения, например:

```
$response->assertSessionHasAll([
    'name' => 'Taylor Otwell',
    'status' => 'active',
]);
```

## assertSessionHasErrors

Утверждает, что сессия содержит ошибку для переданных `$keys`. Если `$keys` является ассоциативным массивом, следует утверждать, что сессия содержит конкретное сообщение об ошибке (значение) для каждого поля (ключа). Этот метод следует использовать при тестировании маршрутов, которые передают ошибки валидации в сессию вместо того, чтобы возвращать их в виде структуры JSON:

```
$response->assertSessionHasErrors(
    array $keys = [], $format = null, $errorBag = 'default'
);
```

Например, чтобы утверждать, что поля `name` и `email` содержат сообщения об ошибках валидации, которые были переданы в сессию, вы можете вызвать метод `assertSessionHasErrors` следующим образом:

```
$response->assertSessionHasErrors(['name', 'email']);
```

Или вы можете утверждать, что переданное поле имеет конкретное сообщение об ошибке валидации:

```
$response->assertSessionHasErrors([
    'name' => 'The given name was invalid.'
]);
```

Более общий метод `assertInvalid` может быть использован для проверки, что ответ содержит ошибки валидации, представленные в формате JSON **или** что ошибки были сохранены в хранилище сессий.

## assertSessionHasErrorsIn

Утверждает, что сессия содержит ошибку для переданных `$keys` в конкретной [коллекции ошибок](#). Если `$keys` является ассоциативным массивом, убедитесь, что сессия содержит конкретное сообщение об ошибке (значение) для каждого поля (ключа) в коллекции ошибок:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

## assertSessionHasNoErrors

Утверждает, что в сессии нет ошибок валидации:

```
$response->assertSessionHasNoErrors();
```

## assertSessionDoesntHaveErrors

Утверждает, что в сессии нет ошибок валидации для переданных ключей:

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag = 'de-
```

Более общий метод [assertValid](#) может быть использован для проверки того, что ответ не содержит ошибок валидации, представленных в формате JSON и что ошибок не было сохранено в хранилище сессий.

## assertSessionMissing

Утверждает, что сессия не содержит переданного ключа:

```
$response->assertSessionMissing($key);
```

## assertStatus

Утверждает, что ответ имеет указанный код `$code` состояния HTTP:

```
$response->assertStatus($code);
```

## assertSuccessful

Утверждает, что ответ имеет код `>= 200` и `< 300` состояния HTTP – `successful`:

```
$response->assertSuccessful();
```

## assertTooManyRequests

Утверждает, что ответ имеет код `429` состояния HTTP – `too many requests`:

```
$response->assertTooManyRequests();
```

## assertUnauthorized

Утверждает, что ответ имеет код `401` состояния HTTP – `unauthorized`:

```
$response->assertUnauthorized();
```

## assertUnprocessable

Утверждает, что ответ имеет необработанный код `422` состояния HTTP:

```
$response->assertUnprocessable();
```

## assertUnsupportedMediaType

Утверждает, что ответ имеет код `415` состояния HTTP – `unsupported media type`:

```
$response->assertUnsupportedMediaType()
```

## assertValid

Утверждает, что в ответе нет ошибок валидации для заданных ключей. Этот метод можно использовать для утверждения против ответов, в которых ошибки проверки возвращаются в виде структуры JSON или ошибки проверки были переданы в сессию:

```
// Assert that no validation errors are present...
$response->assertValid();

// Assert that the given keys do not have validation errors...
$response->assertValid(['name', 'email']);
```

## assertInvalid

Утверждает, что в ответе есть ошибки валидации для заданных ключей. Этот метод можно использовать для утверждения против ответов, где ошибки проверки возвращаются в виде структуры JSON или где ошибки проверки были переданы в сессию:

```
$response->assertInvalid(['name', 'email']);
```

Вы также можете утверждать, что данный ключ имеет определенное сообщение об ошибке валидации. При этом вы можете предоставить все сообщение или только небольшую его часть:

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

## assertViewHas

Утверждает, что шаблон ответа содержит переданный фрагмент данных:

```
$response->assertViewHas($key, $value = null);
```

Передача закрытия в качестве второго аргумента методу `assertViewHas` позволит вам проверять и делать утверждения в отношении определенного фрагмента данных представления:

```
$response->assertViewHas('user', function (User $user) {
    return $user->name === 'Taylor';
});
```

Кроме того, данные шаблона могут быть доступны как переменные массива в ответе, что позволяет вам удобно инспектировать их:

Pest      PHPUnit

```
expect($response['name'])->toBe('Taylor');
```

## assertViewHasAll

Утверждает, что шаблон ответа содержит переданный список данных:

```
$response->assertViewHasAll(array $data);
```

Этот метод может использоваться, чтобы утверждать, что шаблон просто содержит данные с соответствующими переданными ключами:

```
$response->assertViewHasAll([
    'name',
    'email',
]);
```

Или вы можете утверждать, что данные шаблона присутствуют и имеют определенные значения:

```
$response->assertViewHasAll([
    'name' => 'Taylor Otwell',
    'email' => 'taylor@example.com',
]);
```

## assertViewIs

Утверждает, что маршрутом был возвращен указанный шаблон:

```
$response->assertViewIs($value);
```

## assertViewMissing

Утверждает, что переданный ключ данных не был доступен для шаблона, возвращенного ответом приложения:

```
$response->assertViewMissing($key);
```

## Утверждения аутентификации

Laravel также содержит множество утверждений, связанных с аутентификацией, которые вы можете использовать в функциональных тестах вашего приложения. Обратите внимание, что эти методы вызываются в самом тестовом классе, а не в экземпляре [Illuminate\Testing\TestResponse](#), возвращаемом такими методами, как `get` и `post`.

## assertAuthenticated

Утверждает, что пользователь аутентифицирован:

```
$this->assertAuthenticated($guard = null);
```

## assertGuest

Утверждает, что пользователь не аутентифицирован:

```
$this->assertGuest($guard = null);
```

## assertAuthenticatedAs

Утверждает, что конкретный пользователь аутентифицирован:

```
$this->assertAuthenticatedAs($user, $guard = null);
```

## # Утверждения валидации

Laravel предоставляет два основных метода утверждения, связанных с валидацией, которые вы можете использовать, чтобы убедиться, что данные, предоставленные в вашем запросе, являются валидными или невалидными.

### assertValid

Утверждает, что ответ не содержит ошибок валидации для указанных ключей. Этот метод может использоваться для проверки ответов, где ошибки валидации представлены в виде JSON-структуры или где ошибки валидации сохраняются в сессии:

```
// Утверждает, что ошибок валидации нет...
$response->assertValid();

// Утверждает, что нет ошибок валидации для указанных ключей...
$response->assertValid(['name', 'email']);
```

### assertInvalid

Утверждает, что ответ содержит ошибки валидации для указанных ключей. Этот метод может использоваться для проверки ответов, где ошибки валидации представлены в виде JSON-структуры или где ошибки валидации сохраняются в сессии:

```
$response->assertInvalid(['name', 'email']);
```

Также вы можете утверждать, что для указанного ключа есть определенное сообщение об ошибке валидации. При этом вы можете предоставить либо полное сообщение, либо только его небольшую часть:

```
$response->assertInvalid([
    'name' => 'The name field is required.',
```

```
'email' => 'valid email address',  
]);
```

# Тестирование · Тесты консольных команд

- # Введение
- # Ожидания успеха / неудачи
- # Ожидания ввода / вывода
- # События консоли

## # Введение

Помимо упрощенного HTTP-тестирования, Laravel предлагает простой API для тестирования [пользовательских консольных команд](#) вашего приложения.

## # Ожидания успеха / неудачи

Для начала давайте рассмотрим, как делать утверждения относительно кода выхода команды Artisan. Для этого мы будем использовать метод `artisan` для вызова Artisan-команды из нашего теста. Затем мы будем использовать метод `assertExitCode`, чтобы подтвердить, что команда завершилась с заданным кодом выхода:

Pest      PHPUnit

```
test('console command', function () {  
    $this->artisan('inspire')->assertExitCode(0);  
});
```

Вы можете использовать метод `assertNotExitCode` чтобы подтвердить, что команда не завершилась с заданным кодом выхода:

```
$this->artisan('inspire')->assertNotExitCode(1);
```

Конечно, все команды терминала обычно завершаются с кодом состояния `0`, когда они успешны, и с ненулевым кодом выхода, когда они не успешны. Поэтому для удобства вы можете использовать утверждения `assertSuccessful` и `assertFailed` чтобы утверждать, что данная команда завершилась с успешным кодом выхода или нет:

```
$this->artisan('inspire')->assertSuccessful();  
  
$this->artisan('inspire')->assertFailed();
```

## # Ожидания ввода / вывода

Laravel позволяет вам легко «имитировать» ввод пользователем в консольных командах, используя метод `expectsQuestion`. Кроме того, вы можете указать код выхода / возврата и текст, который вы ожидаете получить от консольной команды, используя методы `assertExitCode` и `expectsOutput`. Например, рассмотрим следующую консольную команду:

```
Artisan::command('question', function () {  
    $name = $this->ask('What is your name?');  
  
    $language = $this->choice('Which language do you prefer?', [  
        'PHP',  
        'Ruby',  
        'Python',  
    ]);  
  
    $this->line("Your name is '$name' and you prefer '$language'.");  
});
```

Вы можете проверить эту команду с помощью следующего теста:

Pest      PHPUnit

```
test('console command', function () {  
    $this->artisan('question')  
        ->expectsQuestion('What is your name?', 'Taylor Otwell')  
        ->expectsQuestion('Which language do you prefer?', 'PHP')  
        ->expectsOutput('Your name is Taylor Otwell and you prefer PHP.')  
        ->doesntExpectOutput('Your name is Taylor Otwell and you prefer Ruby.')  
});
```

```
->assertExitCode(0);
});
```

Если вы используете функции `search` или `multisearch`, предоставляемые [Laravel Prompts](#), вы можете использовать утверждение `expectsSearch`, чтобы имитировать ввод пользователя, результаты поиска и выбор:

Pest      PHPUnit

```
test('console command', function () {
    $this->artisan('example')
        ->expectsSearch('What is your name?', search: 'Tay', answers: [
            'Taylor Otwell',
            'Taylor Swift',
            'Darian Taylor'
        ], answer: 'Taylor Otwell')
        ->assertExitCode(0);
});
```

Вы также можете утверждать, что консольная команда не генерирует никакого вывода, используя метод `doesntExpectOutput`:

Pest      PHPUnit

```
test('console command', function () {
    $this->artisan('example')
        ->doesntExpectOutput()
        ->assertExitCode(0);
});
```

The `expectsOutputToContain` and `doesntExpectOutputToContain` methods may be used to make assertions against a portion of the output: Методы `expectsOutputToContain` и `doesntExpectOutputToContain` могут использоваться для создания утверждений относительно части вывода:

Pest      PHPUnit

```
test('console command', function () {
    $this->artisan('example')
        ->expectsOutputToContain('Taylor')
        ->assertExitCode(0);
});
```

## Ожидания подтверждения

При написании команды, которая ожидает подтверждения в виде ответа «да» или «нет», вы можете использовать метод `expectsConfirmation`:

```
$this->artisan('module:import')
    ->expectsConfirmation('Do you really wish to run this command?', 'no')
    ->assertExitCode(1);
```

## Таблица ожиданий

Если ваша команда отображает таблицу информации с использованием метода `table` Artisan, может быть обременительно записывать ожидаемые результаты для всей таблицы. Вместо этого вы можете использовать метод `expectsTable`. Этот метод принимает заголовки таблицы в качестве первого аргумента и данные таблицы в качестве второго аргумента:

```
$this->artisan('users:all')
    ->expectsTable([
        'ID',
        'Email',
    ], [
        [1, 'taylor@example.com'],
        [2, 'abigail@example.com'],
    ]);
```

## # События консоли

По умолчанию события `Illuminate\Console\Events\CommandStarting` и `Illuminate\Console\Events\CommandFinished` не генерируются при запуске тестов вашего приложения. Однако вы можете включить эти события для данного класса тестов, добавив трейт `Illuminate\Foundation\Testing\WithConsoleEvents` в класс:

Pest      PHPUnit

```
<?php
```

```
use Illuminate\Foundation\Testing\WithConsoleEvents;

uses(WithConsoleEvents::class);
```

// ...

# Laravel Dusk

## # Введение

## # Установка

# Управление установками ChromeDriver

# Использование других браузеров

## # Начало работы

# Генерация тестов

# Сброс базы данных после каждого теста

# Запуск тестов

# Обработка файла переменных окружения

## # Основы работы с браузером

# Создание браузеров

# Навигация

# Изменение размера окна браузера

# Макрокоманды браузера

# Аутентификация

# Cookies

# Выполнение JavaScript

# Получение снимка экрана

# Сохранение вывода консоли на диск

# Сохранение исходного кода страницы на диск

## # Взаимодействие с элементами

# Селекторы Dusk

# Текст, значения и атрибуты

# Взаимодействие с формами

# Прикрепление файлов

# Нажатие кнопок

# Клик по ссылкам

# Использование клавиатуры

# Использование мыши

- # Диалоговые окна JavaScript (Alert, Prompt, Confirm)
- # Взаимодействие с фреймами
- # Сегментированное тестирование по селекторам
- # Ожидание доступности элементов
- # Прокрутка элемента в область видимости пользователя

## # Доступные утверждения

### # Тестовые страницы

- # Генерация тестовых страниц
- # Конфигурирование тестовых страниц
- # Навигация по тестовым страницам
- # Псевдонимы селекторов
- # Методы тестовых страниц

### # Компоненты для тестов

- # Генерация компонентов
- # Использование компонентов

### # Непрерывная интеграция

- # Heroku CI
- # Travis CI
- # GitHub Actions
- # Chipper CI

## # Введение

[Laravel Dusk](#) предоставляет выразительный и простой в использовании API для автоматизации и тестирования браузера. По умолчанию Dusk не требует установки JDK или Selenium на ваш локальный компьютер. Вместо этого Dusk использует автономную установку [ChromeDriver](#). По желанию вы можете использовать любой другой драйвер, совместимый с Selenium.

## # Установка

Для начала установите [Google Chrome](#) и `laravel/dusk` с помощью менеджера пакетов Composer в свой проект:

```
composer require laravel/dusk --dev
```

Если вы вручную регистрируете поставщика `DuskServiceProvider`, вам **никогда** не следует регистрировать его в рабочем окружении, так как это может привести к тому, что случайные пользователи смогут пройти аутентификацию в вашем приложении.

После установки пакета Dusk выполните команду Artisan `dusk:install`. Команда `dusk:install` создаст директорию `tests/Browser`, пример теста Dusk и установит двоичный файл Chrome Driver для вашей операционной системы:

```
php artisan dusk:install
```

Затем установите переменную окружения `APP_URL` в файле `.env` вашего приложения. Это значение должно соответствовать URL-адресу, который вы используете для доступа к вашему приложению в браузере.

Если вы используете [Laravel Sail](#) для управления своей локальной средой разработки, то обратитесь также к документации Sail по [настройке и запуску тестов Dusk](#).

## Управление установками ChromeDriver

Если вы хотите установить другую версию ChromeDriver, отличную от той, которая устанавливается Laravel Dusk через команду `dusk:install`, вы можете использовать команду `dusk:chrome-driver`:

```
# Установить последнюю версию ChromeDriver для вашей ОС ...
php artisan dusk:chrome-driver
```

```
# Установить конкретную версию ChromeDriver для вашей ОС ...
php artisan dusk:chrome-driver 86

# Установить конкретную версию ChromeDriver для всех поддерживаемых ОС ...
php artisan dusk:chrome-driver --all

# Установить версию ChromeDriver, которая соответствует обнаруженной версии Chrome /
php artisan dusk:chrome-driver --detect
```

Dusk требует, чтобы файлы `chromedriver` были доступны для выполнения. Если у вас возникли проблемы с запуском Dusk, то вы должны убедиться, что файлы доступны для выполнения, используя следующую команду: `chmod -R 0755 vendor/laravel/dusk/bin/`.

## Использование других браузеров

По умолчанию Dusk использует Google Chrome и автономную установку [ChromeDriver](#) для запуска ваших браузерных тестов. Тем не менее вы можете запустить свой собственный сервер Selenium и запускать тесты в любом браузере по желанию.

Для начала откройте файл `tests/DuskTestCase.php`, который является базовым тестовым классом Dusk вашего приложения. Внутри этого файла вы можете удалить вызов метода `startChromeDriver`. Это остановит Dusk от автоматического запуска ChromeDriver:

```
/**
 * Подготовить Dusk для выполнения теста.
 *
 * @beforeClass
 */
public static function prepare(): void
{
    // static::startChromeDriver();
}
```

Затем вы можете изменить метод `driver` для подключения к URL-адресу и порту по вашему выбору. Кроме того, вы можете изменить «требуемые характеристики» через класс `DesiredCapabilities`, передаваемые экземпляру WebDriver:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;

/**
 * Создать экземпляр RemoteWebDriver.
 */
protected function driver(): RemoteWebDriver
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

## # Начало работы

### Генерация тестов

Чтобы сгенерировать тест Dusk, используйте команду `dusk:make` Artisan.

Сгенерированный тест будет помещен в каталог `tests/Browser`:

```
php artisan dusk:make LoginTest
```

### Сброс базы данных после каждого теста

Большинство тестов, которые вы пишете, будут взаимодействовать со страницами, получающими данные из базы данных вашего приложения; однако, ваши тесты Dusk никогда не должны использовать трейт `RefreshDatabase`. Трейт `RefreshDatabase` использует транзакции базы данных, которые не будут применимы или доступны в течение HTTP-запросов. Вместо этого у вас есть два варианта: трейт `DatabaseMigrations` и трейт `DatabaseTruncation`.

### Использование миграций

Трейт `DatabaseMigrations` будет запускать миграции базы данных перед каждым тестом. Однако удаление и воссоздание таблиц базы данных для каждого теста обычно происходит медленнее, чем очистка таблиц:

Pest      PHPUnit

```
<?php

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;

uses(DatabaseMigrations::class);

//
```

Базы данных SQLite, хранимые в памяти, нельзя использовать при выполнении тестов Dusk. Поскольку браузер выполняет свой собственный процесс, он не сможет получить доступ к базам данных, хранимых в памяти, других процессов.

## Использование Truncation

Трейт `DatabaseTruncation` проведет миграцию вашей базы данных перед первым тестом, чтобы убедиться, что таблицы базы данных были правильно созданы. Однако в последующих тестах таблицы базы данных будут просто очищены, что обеспечивает ускорение по сравнению с повторным выполнением всех миграций базы данных:

Pest      PHPUnit

```
<?php

use Illuminate\Foundation\Testing\DatabaseTruncation;
use Laravel\Dusk\Browser;

uses(DatabaseTruncation::class);

//
```

По умолчанию этот трейт очищает все таблицы, кроме таблицы `migrations`. Если вы хотите настроить таблицы, которые должны быть очищены, вы можете определить свойство `$tablesToTruncate` в вашем тестовом классе:

Если вы используете Pest, вам следует определить свойства или методы базового класса `DuskTestCase` или любого класса, расширяемого вашим тестовым файлом.

```
/**
 * Указывает, какие таблицы должны быть очищены.
 *
 * @var array
 */
protected $tablesToTruncate = ['users'];
```

Кроме того, вы можете определить свойство `$exceptTables` в вашем тестовом классе, чтобы указать, какие таблицы должны быть исключены из очистки:

```
/**
 * Указывает, какие таблицы должны быть исключены из очистки.
 *
 * @var array
 */
protected $exceptTables = ['users'];
```

Чтобы указать, в каких соединениях базы данных должны быть очищены таблицы, вы можете определить свойство `$connectionsToTruncate` в вашем тестовом классе:

```
/**
 * Указывает, в каких соединениях должны быть очищены таблицы.
 *
 * @var array
 */
protected $connectionsToTruncate = ['mysql'];
```

Если вы хотите выполнить код до или после выполнения очистки базы данных, вы можете определить методы `beforeTruncatingDatabase` ИЛИ `afterTruncatingDatabase` в вашем тестовом классе:

```
/**  
 * Выполните любые действия, которые должны быть выполнены перед началом очистки базы  
 */  
protected function beforeTruncatingDatabase(): void  
{  
    //  
}  
  
/**  
 * Выполните любые действия, которые должны быть выполнены после завершения очистки базы  
 */  
protected function afterTruncatingDatabase(): void  
{  
    //  
}
```

## Запуск тестов

Чтобы запустить браузерные тесты, выполните команду `dusk` Artisan:

```
php artisan dusk
```

Если при последнем запуске команды `dusk` у вас были ошибки тестирования, то вы можете сэкономить время, повторно запустив сначала неудачные тесты с помощью команды `dusk:fails`:

```
php artisan dusk:fails
```

Команда `dusk` принимает любой аргумент, который обычно принимается тестером Pest / PHPUnit, например, позволяет вам запускать тесты только для указанной [группы](#):

```
php artisan dusk --group=foo
```

Если вы используете [Laravel Sail](#) для управления своей локальной средой разработки, обратитесь

К документации Sail по [настройке и запуску тестов Dusk](#).

## Запуск ChromeDriver вручную

По умолчанию Dusk автоматически пытается запустить ChromeDriver. Если это не работает для вашей конкретной системы, вы можете вручную запустить ChromeDriver перед запуском команды `dusk`. Если вы решили запустить ChromeDriver вручную, то вы должны закомментировать следующую строку вашего файла `tests/DuskTestCase.php`:

```
/**  
 * Подготовить Dusk для выполнения теста.  
 *  
 * @beforeClass  
 */  
public static function prepare(): void  
{  
    // static::startChromeDriver();  
}
```

Кроме того, если вы запускаете ChromeDriver на порту, отличном от `9515`, то вам следует изменить метод `driver` того же класса, чтобы указать необходимый порт:

```
use Facebook\WebDriver\Remote\RemoteWebDriver;  
  
/**  
 * Создать экземпляр RemoteWebDriver.  
 */  
protected function driver(): RemoteWebDriver  
{  
    return RemoteWebDriver::create(  
        'http://localhost:9515', DesiredCapabilities::chrome()  
    );  
}
```

## Обработка файла переменных окружения

Чтобы заставить Dusk использовать свой собственный файл окружения при запуске тестов, создайте файл `.env.dusk.{environment}` в корне вашего проекта. Например,

если вы будете запускать команду `dusk` из вашей `local` (локальной) среды, то вы должны создать файл `.env.dusk.local`.

При запуске тестов Dusk создаст резервную копию вашего файла `.env` и переименует ваше окружение Dusk в файле `.env`. После завершения тестов ваш файл `.env` будет восстановлен.

## # Основы работы с браузером

### Создание браузеров

Для начала давайте напишем тест, который проверяет, можем ли мы войти в наше приложение. После создания теста мы можем изменить его, чтобы перейти на страницу входа, ввести некоторые учетные данные и нажать кнопку «Войти». Чтобы создать экземпляр браузера, вы можете вызвать метод `browse` из своего теста Dusk:

Pest      PHPUnit

```
<?php

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;

uses(DatabaseMigrations::class);

test('basic example', function () {
    $user = User::factory()->create([
        'email' => 'taylor@laravel.com',
    ]);

    $this->browse(function (Browser $browser) use ($user) {
        $browser->visit('/login')
            ->type('email', $user->email)
            ->type('password', 'password')
            ->press('Login')
            ->assertPathIs('/home');
    });
});
```

Как видно в приведенном выше примере, метод `browse` принимает замыкание. Dusk автоматически передаст экземпляр браузера в это замыкание. Экземпляр браузера является основным объектом, используемым для взаимодействия с вашим приложением и создания утверждений.

## Создание нескольких браузеров

Иногда для правильного проведения теста может потребоваться несколько браузеров. Например, для тестирования экрана чата, взаимодействующего с веб-сокетами, может потребоваться несколько браузеров. Чтобы создать несколько браузеров, просто добавьте больше аргументов браузера к сигнатуре замыкания, передаваемому методу `browse`:

```
$this->browse(function (Browser $first, Browser $second) {  
    $first->loginAs(User::find(1))  
        ->visit('/home')  
        ->waitForText('Message');  
  
    $second->loginAs(User::find(2))  
        ->visit('/home')  
        ->waitForText('Message')  
        ->type('message', 'Hey Taylor')  
        ->press('Send');  
  
    $first->waitForText('Hey Taylor')  
        ->assertSee('Jeffrey Way');  
});
```

## Навигация

Метод `visit` используется для перехода к конкретному URI вашего приложения:

```
$browser->visit('/login');
```

Вы можете использовать метод `visitRoute` для перехода к именованному маршруту:

```
$browser->visitRoute($routeName, $parameters);
```

Вы можете перемещаться «назад» и «вперед», используя методы `back` и `forward`:

```
$browser->back();
```

```
$browser->forward();
```

Вы можете использовать метод `refresh` для обновления страницы:

```
$browser->refresh();
```

## Изменение размера окна браузера

Вы можете использовать метод `resize` для настройки размера окна браузера:

```
$browser->resize(1920, 1080);
```

Метод `maximize` используется для максимизации окна браузера:

```
$browser->maximize();
```

Метод `fitContent` изменит размер окна браузера в соответствии с размером его содержимого:

```
$browser->fitContent();
```

Если тест не пройден, то Dusk автоматически изменяет размер окна браузера в соответствии с его содержимым, прежде чем сделать снимок экрана. Вы можете отключить эту функцию, вызвав в своем тесте метод `disableFitOnFailure`:

```
$browser->disableFitOnFailure();
```

Вы можете использовать метод `move`, чтобы переместить окно браузера в другое место на экране:

```
$browser->move($x = 100, $y = 100);
```

# Макрокоманды браузера

Если вы хотите определить собственный метод браузера, который вы можете повторно использовать в различных ваших тестах, вы можете использовать метод `macro` класса `Browser`. Как правило, этот метод следует вызывать из метода `boot` [поставщика служб](#):

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Регистрация макрокоманд браузера Dusk.
     */
    public function boot(): void
    {
        Browser::macro('scrollToElement', function (string $element = null) {
            $this->script("$(document).animate({ scrollTop: $($element).offset().top })");
        });

        return $this;
    }
}
```

Метод `macro` принимает имя в качестве первого аргумента и замыкание в качестве второго. Замыкание будет выполнено при вызове макрокоманды в качестве метода экземпляра `Browser`:

```
$this->browse(function (Browser $browser) use ($user) {
    $browser->visit('/pay')
        ->scrollToElement('#credit-card-details')
        ->assertSee('Enter Credit Card Details');
});
```

# Аутентификация

Часто вы будете тестировать страницы, требующие аутентификации. Вы можете использовать метод Dusk `loginAs`, чтобы избежать взаимодействия с экраном входа в систему вашего приложения во время каждого теста. Метод `loginAs` принимает первичный ключ аутентифицируемой модели или, непосредственно, экземпляр аутентифицируемой модели:

```
use App\Models\User;

$this->browse(function (Browser $browser) {
    $browser->loginAs(User::find(1))
        ->visit('/home');
});
```

После использования метода `loginAs` сессия пользователя будет поддерживаться для всех тестов, находящихся в файле.

## Cookies

Вы можете использовать метод `cookie` для получения или установления зашифрованного значения cookie. По умолчанию все файлы cookie, созданные Laravel, зашифрованы:

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

Вы можете использовать метод `plainCookie` для получения или установления незашифрованного значения cookie:

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

Вы можете использовать метод `deleteCookie` для удаления конкретного файла cookie:

```
$browser->deleteCookie('name');
```

## Выполнение JavaScript

Вы можете использовать метод `script` для выполнения произвольных выражений JavaScript в браузере:

```
$browser->script('document.documentElement.scrollTop = 0');

$browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);

$output = $browser->script('return window.location.pathname');
```

## Получение снимка экрана

Вы можете использовать метод `screenshot`, чтобы сделать снимок экрана и сохранить его с заданным именем файла. Все скриншоты будут храниться в каталоге `tests/Browser/screenshots`:

```
$browser->screenshot('filename');
```

Метод `responsiveScreenshots` может быть использован для создания серии скриншотов на различных контрольных точках:

```
$browser->responsiveScreenshots('filename');
```

Метод `screenshotElement` можно использовать для создания снимка экрана определенного элемента на странице:

```
$browser->screenshotElement('#selector', 'filename');
```

## Сохранение вывода консоли на диск

Вы можете использовать метод `storeConsoleLog` для записи вывода консоли текущего браузера на диск с заданным именем файла. Вывод консоли будет храниться в каталоге `tests/Browser/console`:

```
$browser->storeConsoleLog('filename');
```

## Сохранение исходного кода страницы на диск

Вы можете использовать метод `storeSource` для записи исходного кода текущей страницы на диск с заданным именем файла. Исходный код страницы будет храниться в каталоге `tests/Browser/source`:

```
$browser->storeSource('filename');
```

## # Взаимодействие с элементами

### Селекторы Dusk

Выбор универсальных селекторов CSS для взаимодействия с элементами – одна из самых сложных частей написания тестов Dusk. Со временем изменения клиентского интерфейса могут привести к тому, что селекторы CSS, подобные приведенным ниже, нарушают ваши тесты:

```
// HTML-разметка ...  
  
<button>Login</button>  
  
// Выполнение теста ...  
  
$browser->click('.login-page .container div > button');
```

Селекторы Dusk позволяют сосредоточиться на написании эффективных тестов, а не на запоминании селекторов CSS. Чтобы определить селектор, добавьте к вашему элементу HTML-атрибут `dusk`. Затем, при взаимодействии с браузером Dusk, добавьте к селектору префикс `@`, чтобы управлять закрепленным элементом в вашем тесте:

```
// HTML-разметка ...

<button dusk="login-button">Login</button>

// Выполнение теста ...

$browser->click('@login-button');
```

Если вам нужно, вы можете настроить HTML-атрибут, который использует селектор Dusk, с помощью метода `selectorHtmlAttribute`. Обычно этот метод следует вызывать из метода `boot` вашего `AppServiceProvider` приложения:

```
use Laravel\Dusk\Dusk;

Dusk::selectorHtmlAttribute('data-dusk');
```

## Текст, значения и атрибуты

### Получение и установка значений

Dusk содержит несколько методов для взаимодействия с текущим значением, отображаемым текстом и атрибутами элементов на странице. Например, чтобы получить «значение» элемента, которое соответствует указанному CSS или Dusk селектору, используйте метод `value`:

```
// Получить значение ...
$value = $browser->value('selector');

// Установить значение ...
$browser->value('selector', 'value');
```

Вы можете использовать метод `inputValue` для получения «значения» элемента ввода, имеющего указанное имя поля:

```
$value = $browser->inputValue('field');
```

### Получение текста

Метод `text` используется для получения отображаемого текста элемента, соответствующий указанному селектору:

```
$text = $browser->text('selector');
```

## Получение атрибутов

Наконец, метод `attribute` может быть использован для получения значения атрибута элемента, соответствующий указанному селектору:

```
$attribute = $browser->attribute('selector', 'value');
```

## Взаимодействие с формами

### Ввод значений

Dusk содержит множество методов для взаимодействия с формами и элементами ввода. Во-первых, давайте взглянем на пример ввода текста в поле:

```
$browser->type('email', 'taylor@laravel.com');
```

Обратите внимание, что, нам не требуется передавать селектор CSS в метод `type`, хотя метод принимает его при необходимости. Если селектор CSS не указан, то Dusk будет искать поле `input` или `textarea` с указанным атрибутом `name`.

Чтобы добавить текст в поле, не очищая его содержимое, вы можете использовать метод `append`:

```
$browser->type('tags', 'foo')
->append('tags', ', bar, baz');
```

Вы можете очистить значение поля с помощью метода `clear`:

```
$browser->clear('email');
```

Вы можете указать Dusk печатать медленно, используя метод `typeSlowly`. По умолчанию Dusk будет делать паузу на `100` миллисекунд между нажатиями клавиш. Чтобы изменить время между нажатиями клавиш, вы можете передать соответствующее количество миллисекунд в качестве третьего аргумента метода:

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');  
  
$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

Вы можете использовать метод `appendSlowly` для медленного добавления текста:

```
$browser->type('tags', 'foo')  
    ->appendSlowly('tags', ', bar, baz');
```

## Выпадающие списки

Чтобы выбрать значение, доступное для выпадающего списка, вы можете использовать метод `select`. Как и метод `type`, метод `select` не требует полного селектора CSS. При передаче значения методу `select` вы должны передать значение параметра `value` вместо отображаемого текста:

```
$browser->select('size', 'Large');
```

Вы можете выбрать случайный вариант, опустив второй аргумент:

```
$browser->select('size');
```

Предоставляя массив в качестве второго аргумента метода `select`, вы можете указать методу на выбор нескольких параметров:

```
$browser->select('categories', ['Art', 'Music']);
```

## Флажки

Чтобы «отметить» флажок, вы можете использовать метод `check`. Как и многие другие методы, связанные с вводом, полный селектор CSS не требуется. Если

совпадение селектора CSS не найдено, то Dusk будет искать флажок с соответствующим атрибутом `name`:

```
$browser->check('terms');
```

Метод `uncheck` используется для «снятия галочки» с флажка:

```
$browser->uncheck('terms');
```

## Радиокнопки

Чтобы «выбрать» вариант из радиокнопок, вы можете использовать метод `radio`. Как и многие другие методы, связанные с вводом, полный селектор CSS не требуется. Если совпадение селектора CSS не найдено, то Dusk будет искать радиокнопку с соответствующими атрибутами `name` и `value`:

```
$browser->radio('size', 'large');
```

## Прикрепление файлов

Метод `attach` используется для прикрепления файла к элементу выбора файлов. Как и многие другие методы, связанные с вводом, полный селектор CSS не требуется. Если совпадение селектора CSS не найдено, то Dusk будет искать элемент выбора файлов с соответствующим атрибутом `name`:

```
$browser->attach('photo', __DIR__ . '/photos/mountains.png');
```

Функционал прикрепления требует, чтобы на вашем сервере было установлено и включено расширение `Zip` PHP.

## Нажатие кнопок

Метод `press` используется для нажатия кнопки на странице. Аргумент, передаваемым методу `press`, может быть либо отображаемый текст кнопки, либо CSS / Dusk селектор:

```
$browser->press('Login');
```

При отправке форм многие приложения отключают кнопку отправки формы после ее нажатия, а затем снова включают кнопку, когда HTTP-запрос отправки формы завершен. Чтобы нажать кнопку и дождаться ее повторного включения, вы можете использовать метод `pressAndwaitFor`:

```
// Нажимаем кнопку и ждем ее активности не более 5 секунд ...
$browser->pressAndwaitFor('Save');

// Нажимаем кнопку и ждем ее активности не более 1 секунды ...
$browser->pressAndwaitFor('Save', 1);
```

## Клик по ссылкам

Чтобы щелкнуть ссылку, вы можете использовать метод `clickLink` экземпляра браузера. Метод `clickLink` щелкнет ссылку с указанным видимым текстом:

```
$browser->clickLink($linkText);
```

Вы можете использовать метод `seeLink`, чтобы определить, видна ли на странице ссылка с указанным видимым текстом:

```
if ($browser->seeLink($linkText)) {
    // ...
}
```

Эти методы взаимодействуют с библиотеками jQuery. Если jQuery недоступен на странице, то Dusk автоматически вставит его на страницу, чтобы он был доступен во время теста.

## Использование клавиатуры

Метод `keys` позволяет передавать более сложные последовательности ввода для указанного элемента, чем это обычно доступно при использовании метода `type`. Например, при вводе значений можно поручить Dusk удерживать клавиши-модификаторы. В этом примере клавиша `shift` будет удерживаться, пока строка «`taylor`» вводится в элемент заданного селектора. После ввода «`taylor`», строка «`swift`» будет вводиться без модификаторов:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

Другой значимый пример использования метода `keys` – это отправка комбинации «горячих клавиш» основному селектору CSS вашего приложения:

```
$browser->keys('.app', ['{command}', 'j']);
```

Все модификаторы клавиш, такие как `{command}` заключены в символы `{}` и соответствуют константам, определенным в классе `Facebook\WebDriver\WebDriverKeys`, который можно [найти на GitHub](#).

## Взаимодействия с клавиатурой

Dusk также предоставляет метод `withKeyboard`, который позволяет гибко выполнять сложные взаимодействия с клавиатурой через класс `Laravel\Dusk\Keyboard`. Класс `Keyboard` предоставляет методы `press`, `release`, `type` и `pause`:

```
use Laravel\Dusk\Keyboard;

$browser->withKeyboard(function (Keyboard $keyboard) {
    $keyboard->press('c')
        ->pause(1000)
        ->release('c')
        ->type(['c', 'e', 'o']);
});
```

## Макросы Клавиатуры

Если вы хотите определить пользовательские взаимодействия с клавиатурой, которые вы можете легко повторно использовать в вашем наборе тестов, вы можете использовать метод `macro`, предоставляемый классом `Keyboard`. Обычно этот метод следует вызывать из метода `boot` [поставщика услуг](#):

```
<?php

namespace App\Providers;

use Facebook\WebDriver\WebDriverKeys;
use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Keyboard;
use Laravel\Dusk\OperatingSystem;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Регистрация макросов браузера Dusk.
     */
    public function boot(): void
    {
        Keyboard::macro('copy', function (string $element = null) {
            $this->type([
                OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTI
            ]);

            return $this;
        });

        Keyboard::macro('paste', function (string $element = null) {
            $this->type([
                OperatingSystem::onMac() ? WebDriverKeys::META : WebDriverKeys::CONTI
            ]);

            return $this;
        });
    }
}
```

Функция `macro` принимает имя в качестве первого аргумента и замыкание в качестве второго. Замыкание макроса будет выполнено при вызове макроса в качестве метода экземпляра `Keyboard`:

```
$browser->click('@textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->copy())
->click('@another-textarea')
->withKeyboard(fn (Keyboard $keyboard) => $keyboard->paste());
```

## Использование мыши

### Клик по элементам

Метод `click` используется для щелчка по элементу с указанным CSS / Dusk селектором:

```
$browser->click('.selector');
```

Метод `clickAtXPath` используется для щелчка по элементу с указанным XPath-выражением:

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

Метод `clickAtPoint` используется для щелчка по самому верхнему элементу в точке с координатой, указанной относительно видимой области браузера:

```
$browser->clickAtPoint($x = 0, $y = 0);
```

Метод `doubleClick` используется для имитации двойного щелчка мыши:

```
$browser->doubleClick();
```

```
$browser->doubleClick('.selector');
```

Метод `rightClick` используется для имитации щелчка правой кнопкой мыши:

```
$browser->rightClick();
```

```
$browser->rightClick('.selector');
```

Метод `clickAndHold` используется для имитации нажатия и удержания кнопки мыши.

Последующий вызов метода `releaseMouse` отменяет это поведение и отпускает кнопку мыши:

```
$browser->clickAndHold('.selector');
```

```
$browser->clickAndHold()  
    ->pause(1000)  
    ->releaseMouse();
```

Метод `controlClick` может быть использован для симуляции события `ctrl+click` в браузере:

```
$browser->controlClick();
```

```
$browser->controlClick('.selector');
```

## Наведение мыши

Метод `mouseover` используется, когда вам нужно навести указатель мыши на элемент с заданным CSS или Dusk селектором:

```
$browser->mouseover('.selector');
```

## Перетаскивания

Метод `drag` используется для перетаскивания элемента с указанным селектором, на другой элемент:

```
$browser->drag('.from-selector', '.to-selector');
```

Или вы можете перетащить элемент в одном направлении:

```
$browser->dragLeft('.selector', $pixels = 10);  
$browser->dragRight('.selector', $pixels = 10);  
$browser->dragUp('.selector', $pixels = 10);  
$browser->dragDown('.selector', $pixels = 10);
```

Наконец, вы можете перетащить элемент с указанным смещением:

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

## Диалоговые окна JavaScript (Alert, Prompt, Confirm)

Dusk содержит различные методы для взаимодействия с диалогами JavaScript. Например, вы можете использовать метод `waitForDialog`, чтобы дождаться появления диалогового окна JavaScript. Этот метод принимает необязательный аргумент, указывающий, сколько секунд ждать до появления диалогового окна:

```
$browser->waitForDialog($seconds = null);
```

Метод `assertDialogOpened` используется для утверждения того, что диалоговое окно было отображено и содержит указанное сообщение:

```
$browser->assertDialogOpened('Dialog message');
```

Если диалоговое окно JavaScript содержит поле ввода, то вы можете использовать метод `typeInDialog`, чтобы ввести значение:

```
$browser->typeInDialog('Hello World');
```

Чтобы закрыть открытое диалоговое окно JavaScript, нажав кнопку «OK», вы можете вызвать метод `acceptDialog`:

```
$browser->acceptDialog();
```

Чтобы закрыть открытое диалоговое окно JavaScript, нажав кнопку «Отмена», вы можете вызвать метод `dismissDialog`:

```
$browser->dismissDialog();
```

## Взаимодействие с фреймами

Если вам нужно взаимодействовать с элементами внутри iframe, вы можете использовать метод `withinFrame`. Все взаимодействия с элементами, происходящие в замыкании, предоставленном методу `withinFrame`, будут ограничены контекстом указанного iframe:

```
$browser->withinFrame('#credit-card-details', function ($browser) {
    $browser->type('input[name="cardnumber"]', '4242424242424242')
        ->type('input[name="exp-date"]', '1224')
        ->type('input[name="cvc"]', '123')
        ->press('Pay');
});
```

## Сегментированное тестирование по селекторам

Иногда требуется выполнить несколько операций, принадлежащих конкретному селектору. Например, вы можете утверждать, что некоторый текст существует только в таблице, а затем щелкнуть кнопку в этой таблице. Для этого можно использовать метод `with`. Все операции, выполняемые в рамках замыкания, переданного методу `with`, будут привязаны к исходному селектору:

```
$browser->with('.table', function (Browser $table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

Иногда требуется выполнить утверждения за пределами текущей области. Вы можете использовать для этого методы `elsewhere` и `elsewhereWhenAvailable`:

```
$browser->with('.table', function (Browser $table) {
    // Текущая область `body .table` ...

    $browser->elsewhere('.page-title', function (Browser $title) {
        // Текущая область `body .page-title` ...
        $title->assertSee('Hello World');
    });
}

$browser->elsewhereWhenAvailable('.page-title', function (Browser $title) {
    // Текущая область `body .page-title` ...
    $title->assertSee('Hello World');
});
```

# Ожидание доступности элементов

При тестировании приложений, широко использующих JavaScript, часто возникает необходимость «подождать», пока не станут доступны определенные элементы или данные, прежде чем приступить к тесту. Dusk сделает это легко. Используя различные методы, вы можете подождать, пока элементы станут видимыми на странице, или даже дождаться, пока указанное выражение JavaScript не станет «истинным».

## Ожидание

Если вам нужно просто приостановить тест на определенное количество миллисекунд, используйте метод `pause`:

```
$browser->pause(1000);
```

Если вам нужно приостановить тест, только если определенное условие является `true`, используйте метод `pauseIf`:

```
$browser->pauseIf(App::environment('production'), 1000);
```

Точно так же, если вам нужно приостановить тест, если определенное условие не является `true`, вы можете использовать метод `pauseUnless`:

```
$browser->pauseUnless(App::environment('testing'), 1000);
```

## Ожидание конкретных селекторов

Метод `waitFor` используется для приостановки выполнения теста до тех пор, пока на странице не отобразится элемент с указанным CSS или Dusk селектором. По умолчанию это приостанавливает тест максимум на пять секунд перед выбросом исключения. При необходимости вы можете передать иной порог тайм-аута в качестве второго аргумента метода:

```
// Ожидание селектора не более пяти секунд ...
$browser->waitFor('.selector');
```

```
// Ожидание селектора максимум одну секунду ...
$browser->waitFor('.selector', 1);
```

Вы также можете подождать, пока элемент с указанным селектором не будет содержать необходимый текст:

```
// Ожидание селектора, содержащего указанный текст, не более пяти секунд ...
$browser->waitForTextIn('.selector', 'Hello World');
```

```
// Ожидание селектора, содержащего указанный текст, не более одной секунды ...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

Вы также можете подождать, пока элемент с указанным селектором не исчезнет со страницы:

```
// Ожидание исчезновения селектора не более пяти секунд ...
$browser->waitUntilMissing('.selector');
```

```
// Ожидание исчезновения селектора не более одной секунды ...
$browser->waitUntilMissing('.selector', 1);
```

Или вы можете подождать, пока элемент, соответствующий данному селектору, не будет включен или отключен:

```
// Ожидание не более пяти секунд, пока селектор не будет включен...
$browser->waitUntilEnabled('.selector');
```

```
// Ожидание не более одной секунды, пока селектор не будет включен...
$browser->waitUntilEnabled('.selector', 1);
```

```
// Ожидание не более пяти секунд, пока селектор не будет выключен...
$browser->waitUntilDisabled('.selector');
```

```
// Ожидание не более одной секунды, пока селектор не будет выключен...
$browser->waitUntilDisabled('.selector', 1);
```

## Сегментированное тестирование при доступности селекторов

Иногда требуется дождаться появления элемента с указанным селектором, а затем взаимодействовать с этим элементом. Например, вы можете подождать, пока не станет доступно модальное окно, а затем нажать кнопку «OK» в модальном окне. Для этого можно использовать метод `whenAvailable`. Все операции с элементами, выполняемые в рамках замыкания, будут привязаны к исходному селектору:

```
$browser->whenAvailable('.modal', function (Browser $modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

## Ожидание видимости текста

Метод `waitForText` используется для ожидания видимости текста на странице:

```
// Ожидание видимости текста максимум пять секунд ...
$browser->waitForText('Hello World');

// Ожидание видимости текста максимум одну секунду ...
$browser->waitForText('Hello World', 1);
```

Вы можете использовать метод `waitUntilMissingText`, чтобы дождаться, пока отображаемый текст не будет удален со страницы:

```
// Ожидание удаления текста не более пяти секунд ...
$browser->waitUntilMissingText('Hello World');

// Ожидание удаления текста не более одной секунды ...
$browser->waitUntilMissingText('Hello World', 1);
```

## Ожидание доступности ссылок

Метод `waitForLink` используется для ожидания появления текста указанной ссылки на странице:

```
// Ожидание видимости ссылки не более пяти секунд ...
$browser->waitForLink('Create');
```

```
// Ожидание видимости ссылки не более одной секунды ...
$browser->waitForLink('Create', 1);
```

## Ожидание Input-элементов

Метод `waitForInput` может быть использован для ожидания, пока данное поле ввода не станет видимым на странице:

```
// Ожидание максимум пять секунд для поля ввода...
$browser->waitForInput($field);
```

```
// Ожидание максимум одну секунду для поля ввода...
$browser->waitForInput($field, 1);
```

## Ожидание пути страницы

При утверждении пути, например, `$browser->assertPathIs('/home')`, утверждение может завершиться ошибкой, если `window.location.pathname` обновляется асинхронно. Вы можете использовать метод `waitForLocation`, чтобы подождать, пока расположение не станет необходимым значением:

```
$browser->waitForLocation('/secret');
```

Метод `waitForLocation` также может использоваться для ожидания, пока текущее местоположение окна не станет полным URL:

```
$browser->waitForLocation('https://example.com/path');
```

Вы также можете дождаться расположения [именованного маршрута](#):

```
$browser->waitForRoute($routeName, $parameters);
```

## Ожидание перезагрузки страницы

Если вам нужно сделать утверждения после перезагрузки страницы, используйте метод `waitForReload`:

```
use Laravel\\Dusk\\Browser;

$browser->waitForReload(function (Browser $browser) {
    $browser->press('Submit');
})
->assertSee('Success!');
```

Поскольку необходимость дождаться перезагрузки страницы обычно возникает после нажатия кнопки, вы можете использовать метод `clickAndwaitForReload` для удобства:

```
$browser->clickAndwaitForReload('.selector')
->assertSee('something');
```

## Ожидание выражений JavaScript

По желанию можно приостановить выполнение теста до тех пор, пока указанное выражение JavaScript не станет истинным. Вы можете легко сделать это, используя метод `waitFor`. При передаче выражения в этот метод вам не нужно включать в него ни ключевое слово `return`, ни конечную точку с запятой:

```
// Ожидание истинности выражения не более пяти секунд ...
$browser->waitFor('App.data.servers.length > 0');

// Ожидание истинности выражения не более одной секунды ...
$browser->waitFor('App.data.servers.length > 0', 1);
```

## Ожидание выражений Vue

Методы `waitForVue` и `waitForVueIsNot` могут использоваться для ожидания, пока атрибут `компонента Vue` не получит указанное значение:

```
// Ожидание соответствия атрибута компонента указанному значению ...
$browser->waitForVue('user.name', 'Taylor', '@user');

// Ожидание несоответствия атрибута компонента указанному значению ...
$browser->waitForVueIsNot('user.name', null, '@user');
```

## Ожидание событий JavaScript

Метод `waitForEvent` можно использовать для приостановки выполнения теста до тех пор, пока не произойдет событие JavaScript:

```
$browser->waitForEvent('load');
```

Слушатель событий прикрепляется к текущей области видимости, которая по умолчанию является элементом `body`. При использовании селектора с ограничением области видимости слушатель событий будет прикреплен к соответствующему элементу:

```
$browser->with('iframe', function (Browser $iframe) {
    // Ожидание события загрузки iframe...
    $iframe->waitForEvent('load');
});
```

Вы также можете предоставить селектор в качестве второго аргумента метода `waitForEvent`, чтобы прикрепить слушатель событий к определенному элементу:

```
$browser->waitForEvent('load', '.selector');
```

Вы также можете ожидать событий на объектах `document` и `window`:

```
// Ожидание, пока документ не будет прокручен...
$browser->waitForEvent('scroll', 'document');

// Ожидание максимум пять секунд, пока окно не изменит размер...
$browser->waitForEvent('resize', 'window', 5);
```

## Использование замыканий при ожидании

Многие из методов «ожидания» в Dusk основаны на методе `waitUsing`. Вы можете использовать этот метод напрямую, чтобы дождаться, пока переданное замыкание не вернет `true`. Метод `waitUsing` принимает максимальное количество секунд ожидания, интервал между выполнениями замыкания (паузу), само замыкание и необязательное сообщение об ошибке:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
```

```
}, "Something wasn't ready in time.");
```

## Прокрутка элемента в область видимости пользователя

Иногда вы не можете щелкнуть элемент, потому что он находится за пределами области просмотра браузера. Метод `scrollIntoView` будет прокручивать окно браузера до тех пор, пока элемент с указанным селектором не окажется видимым:

```
$browser->scrollIntoView('.selector')
->click('.selector');
```

## # Доступные утверждения

Dusk содержит множество утверждений, которые вы можете использовать при тестировании вашего приложения. Все доступные утверждения представлены в списке ниже:

[assertTitle](#)  
[assertTitleContains](#)  
[assertUrls](#)  
[assertSchemes](#)  
[assertSchemesNot](#)  
[assertHosts](#)  
[assertHostsNot](#)  
[assertPortIs](#)  
[assertPortIsNot](#)  
[assertPathBeginsWith](#)  
[assertPathEndsWith](#)  
[assertPathContains](#)  
[assertPathIs](#)  
[assertPathIsNot](#)  
[assertRoutes](#)  
[assertQueryStringHas](#)  
[assertQueryStringMissing](#)  
[assertFragments](#)

[assertFragmentBeginsWith](#)  
[assertFragmentsNot](#)  
[assertHasCookie](#)  
[assertHasPlainCookie](#)  
[assertCookieMissing](#)  
[assertPlainCookieMissing](#)  
[assertCookieValue](#)  
[assertPlainCookieValue](#)  
[assertSee](#)  
[assertDontSee](#)  
[assertSeeIn](#)  
[assertDontSeeIn](#)  
[assertSeeAnythingIn](#)  
[assertSeeNothingIn](#)  
[assertScript](#)  
[assertSourceHas](#)  
[assertSourceMissing](#)  
[assertSeeLink](#)  
[assertDontSeeLink](#)  
[assertInputValue](#)  
[assertInputValuesNot](#)  
[assertChecked](#)  
[assertNotChecked](#)  
[assertIndeterminate](#)  
[assertRadioSelected](#)  
[assertRadioNotSelected](#)  
[assertSelected](#)  
[assertNotSelected](#)  
[assertSelectHasOptions](#)  
[assertSelectMissingOptions](#)  
[assertSelectHasOption](#)  
[assertSelectMissingOption](#)  
[assertValue](#)  
[assertValuesNot](#)  
[assertAttribute](#)  
[assertAttributeContains](#)

[assertAttributeDoesntContain](#)

[assertAriaAttribute](#)

[assertDataAttribute](#)

[assertVisible](#)

[assertPresent](#)

[assertNotPresent](#)

[assertMissing](#)

[assertInputPresent](#)

[assertInputMissing](#)

[assertDialogOpened](#)

[assertEnabled](#)

[assertDisabled](#)

[assertButtonEnabled](#)

[assertButtonDisabled](#)

[assertFocused](#)

[assertNotFocused](#)

[assertAuthenticated](#)

[assertGuest](#)

[assertAuthenticatedAs](#)

[assertVue](#)

[assertVuelsNot](#)

[assertVueContains](#)

[assertVueDoesntContain](#)

## assertTitle

Утверждает, что заголовок страницы соответствует переданному тексту:

```
$browser->assertTitle($title);
```

## assertTitleContains

Утверждает, что заголовок страницы содержит переданный текст:

```
$browser->assertTitleContains($title);
```

## **assertUrlIs**

Утверждает, что текущий URL (без строки запроса) соответствует переданной строке:

```
$browser->assertUrlIs($url);
```

## **assertSchemeIs**

Утверждает, что схема текущего URL соответствует переданной схеме:

```
$browser->assertSchemeIs($scheme);
```

## **assertSchemeIsNot**

Утверждает, что схема текущего URL не соответствует переданной схеме:

```
$browser->assertSchemeIsNot($scheme);
```

## **assertHostIs**

Утверждает, что хост текущего URL соответствует переданному хосту:

```
$browser->assertHostIs($host);
```

## **assertHostIsNot**

Утверждает, что хост текущего URL не соответствует переданному хосту:

```
$browser->assertHostIsNot($host);
```

## **assertPortIs**

Утверждает, что порт текущего URL соответствует переданному порту:

```
$browser->assertPortIs($port);
```

## assertPortIsNot

Утверждает, что порт текущего URL не соответствует переданному порту:

```
$browser->assertPortIsNot($port);
```

## assertPathBeginsWith

Утверждает, что путь текущего URL начинается с указанного пути:

```
$browser->assertPathBeginsWith('/home');
```

## assertPathEndsWith

Утверждает, что текущий путь URL-адреса заканчивается заданным путем:

```
$browser->assertPathEndsWith('/home');
```

## assertPathContains

Утверждает, что текущий путь URL-адреса содержит заданный путь:

```
$browser->assertPathContains('/home');
```

## assertPathIs

Утверждает, что текущий путь соответствует переданному пути:

```
$browser->assertPathIs('/home');
```

## assertPathIsNot

Утверждает, что текущий путь не соответствует переданному пути:

```
$browser->assertPathIsNot( '/home' );
```

## assertRouteIs

Утверждает, что текущий URL соответствует переданному URL [именованного маршрута](#):

```
$browser->assertRouteIs($name, $parameters);
```

## assertQueryStringHas

Утверждает, что переданный параметр строки запроса присутствует:

```
$browser->assertQueryStringHas($name);
```

Утверждает, что переданный параметр строки запроса присутствует и имеет указанное значение:

```
$browser->assertQueryStringHas($name, $value);
```

## assertQueryStringMissing

Утверждает, что переданный параметр строки запроса отсутствует:

```
$browser->assertQueryStringMissing($name);
```

## assertFragments

Утверждает, что хеш-фрагмент текущего URL соответствует переданному фрагменту:

```
$browser->assertFragmentIs( 'anchor' );
```

## assertFragmentBeginsWith

Утверждает, что хеш-фрагмент текущего URL начинается с указанного фрагмента:

```
$browser->assertFragmentBeginsWith('anchor');
```

## assertFragmentIsNot

Утверждает, что хеш-фрагмент текущего URL не соответствует переданному фрагменту:

```
$browser->assertFragmentIsNot('anchor');
```

## assertHasCookie

Утверждает, что переданный зашифрованный файл cookie присутствует:

```
$browser->assertHasCookie($name);
```

## assertHasPlainCookie

Утверждает, что переданный незашифрованный файл cookie присутствует:

```
$browser->assertHasPlainCookie($name);
```

## assertCookieMissing

Утверждает, что переданный зашифрованный файл cookie отсутствует:

```
$browser->assertCookieMissing($name);
```

## assertPlainCookieMissing

Утверждает, что переданный незашифрованный файл cookie отсутствует:

```
$browser->assertPlainCookieMissing($name);
```

## **assertCookieValue**

Утверждает, что зашифрованный файл cookie имеет указанное значение:

```
$browser->assertCookieValue($name, $value);
```

## **assertPlainCookieValue**

Утверждает, что незашифрованный файл cookie имеет указанное значение:

```
$browser->assertPlainCookieValue($name, $value);
```

## **assertSee**

Утверждает, что переданный текст присутствует на странице:

```
$browser->assertSee($text);
```

## **assertDontSee**

Утверждает, что переданный текст отсутствует на странице:

```
$browser->assertDontSee($text);
```

## **assertSeeIn**

Утверждает, что переданный текст присутствует в селекторе:

```
$browser->assertSeeIn($selector, $text);
```

## **assertDontSeeIn**

Утверждает, что переданный текст отсутствует в селекторе:

```
$browser->assertDontSeeIn($selector, $text);
```

## **assertSeeAnythingIn**

Утверждает, что в селекторе присутствует какой-либо текст:

```
$browser->assertSeeAnythingIn($selector);
```

## **assertSeeNothingIn**

Утверждает, что в селекторе отсутствует какой-либо текст:

```
$browser->assertSeeNothingIn($selector);
```

## **assertScript**

Утверждает, что переданное выражение JavaScript возвращает указанное либо истинное значение:

```
$browser->assertScript('window.isLoaded')
    ->assertScript('document.readyState', 'complete');
```

## **assertSourceHas**

Утверждает, что переданный исходный код присутствует на странице:

```
$browser->assertSourceHas($code);
```

## **assertSourceMissing**

Утверждает, что переданный исходный код отсутствует на странице:

```
$browser->assertSourceMissing($code);
```

## **assertSeeLink**

Утверждает, что переданная ссылка присутствует на странице:

```
$browser->assertSeeLink($linkText);
```

## assertDontSeeLink

Утверждает, что переданная ссылка отсутствует на странице:

```
$browser->assertDontSeeLink($linkText);
```

## assertInputValue

Утверждает, что переданное поле ввода имеет указанное значение:

```
$browser->assertInputValue($field, $value);
```

## assertInputValueIsNot

Утверждает, что переданное поле ввода не имеет указанное значение:

```
$browser->assertInputValueIsNot($field, $value);
```

## assertChecked

Утверждает, что переданный флажок отмечен:

```
$browser->assertChecked($field);
```

## assertNotChecked

Утверждает, что переданный флажок не отмечен:

```
$browser->assertNotChecked($field);
```

## assertIndeterminate

Утверждение, что данный флажок (checkbox) находится в неопределенном состоянии:

```
$browser->assertIndeterminate($field);
```

## assertRadioSelected

Утверждает, что переданная радиокнопка выбрана:

```
$browser->assertRadioSelected($field, $value);
```

## assertRadioNotSelected

Утверждает, что переданная радиокнопка не выбрана:

```
$browser->assertRadioNotSelected($field, $value);
```

## assertSelected

Утверждает, что в переданном выпадающем списке выбрано указанное значение:

```
$browser->assertSelected($field, $value);
```

## assertNotSelected

Утверждает, что в переданном выпадающем списке не выбрано указанное значение:

```
$browser->assertNotSelected($field, $value);
```

## assertSelectHasOptions

Утверждает, что переданный массив значений доступен для выбора:

```
$browser->assertSelectHasOptions($field, $values);
```

## **assertSelectMissingOptions**

Утверждает, что переданный массив значений недоступен для выбора:

```
$browser->assertSelectMissingOptions($field, $values);
```

## **assertSelectHasOption**

Утверждает, что переданное значение доступно для выбора в указанном поле:

```
$browser->assertSelectHasOption($field, $value);
```

## **assertSelectMissingOption**

Утверждает, что переданное значение недоступно для выбора в указанном поле:

```
$browser->assertSelectMissingOption($field, $value);
```

## **assertValue**

Утверждает, что элемент с указанным селектором, имеет переданное значение:

```
$browser->assertValue($selector, $value);
```

## **assertValueIsNot**

Утверждают, что элемент, соответствующий данному селектору, не имеет заданного значения:

```
$browser->assertValueIsNot($selector, $value);
```

## **assertAttribute**

Утверждает, что элемент с указанным селектором, имеет переданное значение атрибута:

```
$browser->assertAttribute($selector, $attribute, $value);
```

## assertAttributeContains

Утверждают, что элемент, соответствующий данному селектору, содержит заданное значение в предоставленном атрибуте:

```
$browser->assertAttributeContains($selector, $attribute, $value);
```

## assertAttributeDoesntContain

Утверждение, что элемент, соответствующий данному селектору, не содержит заданное значение в указанном атрибуте:

```
$browser->assertAttributeDoesntContain($selector, $attribute, $value);
```

## assertAriaAttribute

Утверждает, что элемент с указанным селектором, имеет переданное значение **aria**-атрибута:

```
$browser->assertAriaAttribute($selector, $attribute, $value);
```

Например, учитывая разметку `<button aria-label="Add"></button>`, вы можете выстроить утверждение относительно атрибута **aria-label** следующим образом:

```
$browser->assertAriaAttribute('button', 'label', 'Add')
```

## assertDataAttribute

Утверждает, что элемент с указанным селектором, имеет переданное значение **data**-атрибута:

```
$browser->assertDataAttribute($selector, $attribute, $value);
```

Например, учитывая разметку `<tr id="row-1" data-content="attendees"></tr>`, вы можете выстроить утверждение относительно атрибута `data-content` следующим образом:

```
$browser->assertDataAttribute('#row-1', 'content', 'attendees')
```

## assertVisible

Утверждает, что элемент с указанным селектором, видим:

```
$browser->assertVisible($selector);
```

## assertPresent

Утверждает, что элемент с указанным селектором, присутствует в исходном коде страницы:

```
$browser->assertPresent($selector);
```

## assertNotPresent

Утверждает, что элемент с указанным селектором, отсутствует в исходном коде страницы:

```
$browser->assertNotPresent($selector);
```

## assertMissing

Утверждает, что элемент с указанным селектором, не виден:

```
$browser->assertMissing($selector);
```

## assertInputPresent

Утверждают, что присутствует "input" с заданным именем:

```
$browser->assertInputPresent($name);
```

## assertInputMissing

Утверждают, что “input” с данным именем отсутствует в источнике:

```
$browser->assertInputMissing($name);
```

## assertDialogOpened

Утверждает, что был открыт диалог JavaScript с указанным сообщением:

```
$browser->assertDialogOpened($message);
```

## assertEnabled

Утверждает, что переданное поле доступно для использования:

```
$browser->assertEnabled($field);
```

## assertDisabled

Утверждает, что переданное поле недоступно для использования:

```
$browser->assertDisabled($field);
```

## assertButtonEnabled

Утверждает, что переданная кнопка доступна для использования:

```
$browser->assertButtonEnabled($button);
```

## assertButtonDisabled

Утверждает, что переданная кнопка недоступна для использования:

```
$browser->assertButtonDisabled($button);
```

## assertFocused

Утверждает, что переданное поле находится в фокусе:

```
$browser->assertFocused($field);
```

## assertNotFocused

Утверждает, что переданное поле не находится в фокусе:

```
$browser->assertNotFocused($field);
```

## assertAuthenticated

Утверждает, что пользователь аутентифицирован:

```
$browser->assertAuthenticated();
```

## assertGuest

Утверждает, что пользователь не аутентифицирован:

```
$browser->assertGuest();
```

## assertAuthenticatedAs

Утверждает, что пользователь аутентифицирован как указанный пользователь:

```
$browser->assertAuthenticatedAs($user);
```

## assertVue

Dusk даже позволяет вам делать утверждения о состоянии данных [компоненты Vue](#). Например, представьте, что ваше приложение содержит следующий компонент Vue:

```
// HTML-разметка ...

<profile dusk="profile-component"></profile>

// Определение компонента ...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
                name: 'Taylor'
            }
        };
    }
});
```

Вы можете утверждать о состоянии компонента Vue следующим образом:

Pest      PHPUnit

```
test('vue', function () {
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
            ->assertVue('user.name', 'Taylor', '@profile-component');
    });
});
```

## assertVueIsNot

Утверждает, что переданное свойство данных компонента Vue не соответствует указанному значению:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

## assertVueContains

Утверждает, что переданное свойство данных компонента Vue является массивом и содержит указанное значение:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

## assertVueDoesntContain

Утверждает, что переданное свойство данных компонента Vue является массивом и не содержит указанное значения:

```
$browser->assertVueDoesntContain($property, $value, $componentSelector = null);
```

## # Тестовые страницы

Иногда тесты требуют последовательного выполнения нескольких сложных действий. Это может затруднить чтение и понимание ваших тестов. Страницы Dusk позволяют вам выразительно определять действия, которые затем могут быть выполнены на данной странице с помощью одного метода. Страницы также позволяют вам определять псевдонимы для общих селекторов всего приложения или отдельной страницы.

## Генерация тестовых страниц

Чтобы сгенерировать класс страницы, выполните команду `dusk:page` Artisan. Все классы страниц будут помещены в каталог `tests/Browser/Pages` вашего приложения:

```
php artisan dusk:page Login
```

## Конфигурирование тестовых страниц

По умолчанию страницы имеют три метода: `url`, `assert` и `elements`. Сейчас мы обсудим методы `url` и `assert`. Метод `elements` будет [более подробно описан ниже](#).

### Метод url

Метод `url` должен возвращать путь URL-адреса, представляющего страницу. Dusk будет использовать этот URL-адрес при переходе на страницу в браузере:

```
/**  
 * Получить URL-адрес страницы.  
 */  
public function url(): string  
{  
    return '/login';  
}
```

## Метод assert

Метод `assert` может делать любые утверждения, необходимые для подтверждения того, что браузер действительно находится на данной странице. На самом деле нет необходимости размещать что-либо в этом методе; однако вы можете сделать эти утверждения, если хотите. Эти утверждения будут запускаться автоматически при переходе на страницу:

```
/**  
 * Подтвердить, что браузер находится на странице.  
 */  
public function assert(Browser $browser): void  
{  
    $browser->assertPathIs($this->url());  
}
```

## Навигация по тестовым страницам

После того как страница определена, вы можете посетить ее с помощью метода `visit`:

```
use Tests\Browser\Pages\Login;  
  
$browser->visit(new Login);
```

Иногда, уже находясь на какой-либо странице, вам необходимо «загрузить» селекторы и методы страницы в текущий контекст теста. Это обычное явление, когда вы нажимаете кнопку и перенаправляйтесь на указанную страницу без

явного перехода к ней. В этой ситуации вы можете использовать метод `on` для загрузки страницы:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

## Псевдонимы селекторов

Метод `elements` внутри классов страниц позволяет вам определять быстрые, легко запоминающиеся псевдонимы для любого селектора CSS на вашей странице. Например, давайте определим псевдоним для поля ввода «электронная почта» на странице входа в приложение:

```
/**
 * Получить псевдонимы элементов страницы.
 *
 * @return array<string, string>
 */
public function elements(): array
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

После того как псевдоним был определен, вы можете использовать сокращенный селектор в любом месте, где вы обычно используете полный селектор CSS:

```
$browser->type('@email', 'taylor@laravel.com');
```

## Глобальные псевдонимы селекторов

После установки Dusk базовый класс `Page` будет помещен в ваш каталог `tests/Browser/Pages`. Этот класс содержит метод `siteElements`, используемый для определения глобальных псевдонимов селекторов, которые должны быть доступны на каждой странице вашего приложения:

```

/**
 * Получить глобальные псевдонимы элементов сайта.
 *
 * @return array<string, string>
 */
public static function siteElements(): array
{
    return [
        '@element' => '#selector',
    ];
}

```

## Методы тестовых страниц

В дополнение к методам по умолчанию, определенным на тестовых страницах, вы можете определить дополнительные методы, которые могут использоваться в ваших тестах. Например, представим, что мы создаем приложение для управления музыкой. Обычным действием для одной страницы приложения может быть создание списка воспроизведения. Вместо того чтобы переписывать логику создания списка воспроизведения в каждом тесте, вы можете определить пользовательский метод `createPlaylist` в классе страницы:

```

<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Page;

class Dashboard extends Page
{
    // Другие методы страницы ...

    /**
     * Создать новый список воспроизведения.
     *
     * @param \Laravel\Dusk\Browser $browser
     * @param string $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}

```

```
    }  
}
```

Как только метод определен, вы можете использовать его в любом тесте, использующем данную страницу. Экземпляр браузера будет автоматически внедрен в качестве первого аргумента пользовательским методам страницы:

```
use Tests\Browser\Pages\Dashboard;  
  
$browser->visit(new Dashboard)  
    ->createPlaylist('My Playlist')  
    ->assertSee('My Playlist');
```

## # Компоненты для тестов

Компоненты похожи на «классы страниц» Dusk, но предназначены для частей пользовательского интерфейса и функций, которые повторно используются в вашем приложении, таких как панель навигации или окно уведомлений. Таким образом, компоненты не привязаны к конкретным URL-адресам.

## Генерация компонентов

Чтобы сгенерировать компонент, выполните команду `dusk:component` Artisan. Новые компоненты будут помещены в каталог `tests/Browser/Components`:

```
php artisan dusk:component DatePicker
```

Компонент «выбора даты» является примером компонента, который может присутствовать в вашем приложении на различных страницах. Может оказаться обременительным вручную написать логику автоматизации браузера для выбора даты в десятках тестов. Вместо этого мы можем определить компонент Dusk для представления элемента выбора даты, что позволит нам инкапсулировать эту логику внутри компонента:

```
<?php  
  
namespace Tests\Browser\Components;
```

```
use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Получить корневой селектор компонента.
     */
    public function selector(): string
    {
        return '.date-picker';
    }

    /**
     * Подтвердить, что страница браузера содержит компонент.
     */
    public function assert(Browser $browser): void
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Получить псевдонимы элементов компонента.
     *
     * @return array<string, string>
     */
    public function elements(): array
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Выбрать дату.
     */
    public function selectDate(Browser $browser, int $year, int $month, int $day): void
    {
        $browser->click('@date-field')
            ->within('@year-list', function (Browser $browser) use ($year) {
                $browser->click($year);
            })
            ->within('@month-list', function (Browser $browser) use ($month) {
                $browser->click($month);
            })
            ->within('@day-list', function (Browser $browser) use ($day) {

```

```
        $browser->click($day);
    });
}
}
```

## Использование компонентов

Как только компонент определен, мы можем легко выбрать дату с помощью элемента выбора даты из любого теста. И, если логика, необходимая для выбора даты, изменится, нам нужно будет только обновить компонент:

Pest      PHPUnit

```
<?php

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;

uses(DatabaseMigrations::class);

test('basic example', function () {
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
            ->within(new DatePicker, function (Browser $browser) {
                $browser->selectDate(2019, 1, 30);
            })
            ->assertSee('January');
    });
});
```

## # Непрерывная интеграция

Большинство конфигураций непрерывной интеграции Dusk предполагают, что ваше приложение Laravel будет обслуживаться с помощью встроенного сервера разработки PHP на порту **8000**. Поэтому, прежде чем продолжить, вы должны убедиться, что ваша среда непрерывной

интеграции имеет значение переменной окружения `APP_URL`, равное `http://127.0.0.1:8000`.

## Heroku CI

Чтобы запустить тесты Dusk на [Heroku CI](#), добавьте следующий пакет сборки и скрипты Google Chrome в свой файл `app.json` Heroku:

```
{  
  "environments": {  
    "test": {  
      "buildpacks": [  
        { "url": "heroku/php" },  
        { "url": "https://github.com/heroku/heroku-buildpack-chrome-for-testing" }  
      ],  
      "scripts": {  
        "test-setup": "cp .env.testing .env",  
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux --port='  
      }  
    }  
  }  
}
```

## Travis CI

Чтобы запустить тесты Dusk на [Travis CI](#), используйте следующую конфигурацию `.travis.yml`. Поскольку Travis CI не является графической средой, то нам нужно будет предпринять некоторые дополнительные шаги, чтобы запустить браузер Chrome. Кроме того, мы будем использовать `php artisan serve` для запуска встроенного веб-сервера PHP:

```
language: php  
  
php:  
  - 8.2  
  
addons:  
  chrome: stable  
  
install:  
  - cp .env.testing .env
```

```
- travis_retry composer install --no-interaction --prefer-dist
- php artisan key:generate
- php artisan dusk:chrome-driver

before_script:
- google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://
- php artisan serve --no-reload &

script:
- php artisan dusk
```

## GitHub Actions

Если вы используете [Github Actions](#) для запуска тестов Dusk, то вы можете использовать следующий конфигурационный файл в качестве отправной точки. Как и в случае с TravisCI, мы будем использовать команду `php artisan serve` для запуска встроенного веб-сервера PHP:

```
name: CI
on: [push]
jobs:
  dusk-php:
    runs-on: ubuntu-latest
    env:
      APP_URL: "http://127.0.0.1:8000"
      DB_USERNAME: root
      DB_PASSWORD: root
      MAIL_MAILER: log
    steps:
      - uses: actions/checkout@v4
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: |
          sudo systemctl start mysql
          mysql --user="root" --password="root" -e "CREATE DATABASE `my-database`"
      - name: Install Composer Dependencies
        run: composer install --no-progress --prefer-dist --optimize-autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver --detect
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux --port=9515 &
```

```
- name: Run Laravel Server
  run: php artisan serve --no-reload &
- name: Run Dusk Tests
  run: php artisan dusk
- name: Upload Screenshots
  if: failure()
  uses: actions/upload-artifact@v4
  with:
    name: screenshots
    path: tests/Browser/screenshots
- name: Upload Console Logs
  if: failure()
  uses: actions/upload-artifact@v4
  with:
    name: console
    path: tests/Browser/console
```

## Chipper CI

Если вы используете [Chipper CI](#) для запуска ваших тестов Dusk, вы можете использовать следующий конфигурационный файл в качестве отправной точки. Мы будем использовать встроенный сервер PHP для запуска Laravel, чтобы прослушивать запросы:

```
# файл .chipperci.yml
version: 1

environment:
  php: 8.2
  node: 16

# Включаем Chrome в среду сборки
services:
  - dusk

# Собираем все коммиты
on:
  push:
    branches: .*

pipeline:
  - name: Setup
    cmd: |
      cp -v .env.example .env
      composer install --no-interaction --prefer-dist --optimize-autoloader
```

```
php artisan key:generate

# Создаем файл окружения dusk, убедившись, что APP_URL использует BUILD_HOST
cp -v .env .env.dusk.ci
sed -i "s@APP_URL=.*@APP_URL=http://$BUILD_HOST:8000@g" .env.dusk.ci

- name: Compile Assets
  cmd: |
    npm ci --no-audit
    npm run build

- name: Browser Tests
  cmd: |
    php -S [::0]:8000 -t public 2>server.log &
    sleep 2
    php artisan dusk:chrome-driver $CHROME_DRIVER
    php artisan dusk --env=ci
```

Чтобы узнать больше о запуске тестов Dusk на Chipper CI, включая использование баз данных, ознакомьтесь с [официальной документацией Chipper CI](#).

# Тестирование · База данных

## # Введение

# Сброс базы данных после каждого теста

## # Фабрики моделей

## # Запуск наполнителей (seed, seeders)

## # Доступные утверждения

## # Введение

Laravel предлагает множество полезных инструментов, чтобы упростить тестирование приложений, использующих базу данных. Фабрики моделей (factory) и наполнители (seeders) позволяют безболезненно создавать записи тестовой базы данных с использованием моделей и отношений Eloquent вашего приложения. Мы обсудим все эти мощные функции в текущей документации.

## Сброс базы данных после каждого теста

Прежде чем продолжить, давайте обсудим, как сбрасывать вашу базу данных после каждого из ваших тестов, чтобы данные из предыдущего теста не мешали последующим тестам. Включенный в Laravel трейт

[Illuminate\Foundation\Testing\RefreshDatabase](#) позаботится об этом за вас. Просто используйте трейт в своем тестовом классе:

Pest      PHPUnit

```
<?php

use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('basic example', function () {
    $response = $this->get('/');
    expect($response->status())
        ->toBe(200);
})
```

```
// ...
});
```

Трейт `Illuminate\Foundation\Testing\RefreshDatabase` не мигрирует вашу базу данных, если ваша схема актуальна. Вместо этого он выполняет тест в пределах транзакции базы данных. Следовательно, любые записи, добавленные в базу данных в тестах, не использующих этот трейт, могут по-прежнему существовать в базе данных.

## # Фабрики моделей

При тестировании может возникнуть необходимость вставить несколько записей в вашу базу данных перед выполнением теста. Вместо ручного указания значения каждого столбца при создании тестовых данных Laravel позволяет вам определить набор атрибутов по умолчанию для каждой [модели Eloquent](#), используя [фабрики моделей](#).

Для более подробной информации о создании и использовании фабрик моделей для создания моделей обратитесь к полной [документации по фабрикам моделей](#). После того, как вы определили фабрику модели, вы можете использовать ее внутри вашего теста для создания моделей:

Pest      PHPUnit

```
use App\Models\User;

test('models can be instantiated', function () {
    $user = User::factory()->create();

    // ...
});
```

## # Запуск наполнителей (seed, seeders)

Если вы хотите использовать [наполнители базы данных](#) для заполнения вашей базы данных во время функционального тестирования, то вы можете вызвать метод `seed`. По умолчанию метод `seed` будет запускать `DatabaseSeeder`, который должен запускать все другие ваши наполнители. Как вариант, вы можете передать конкретное имя класса-наполнителя методу `seed`:

Pest      PHPUnit

```
<?php

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

test('orders can be created', function () {
    // Run the DatabaseSeeder...
    $this->seed();

    // Run a specific seeder...
    $this->seed(OrderStatusSeeder::class);

    // ...

    // Run an array of specific seeders...
    $this->seed([
        OrderStatusSeeder::class,
        TransactionStatusSeeder::class,
        // ...
    ]);
});
```

В качестве альтернативы, вы можете указать Laravel автоматически заполнять базу данных перед каждым тестом, который использует трейт `RefreshDatabase`. Вы можете добиться этого, определив свойство `$seed` в вашем базовом тестовом классе:

```
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    /**
     * Указывает, следует ли запускать наполнитель по умолчанию перед каждым тестом.
     *
     * @var bool
     */
    protected $seed = true;
```

```
}
```

Когда свойство `$seed` имеет значение `true`, тогда класс `Database\Seeders\DatabaseSeeder` будет запускаться перед каждым тестом, который использует трейт `RefreshDatabase`. Однако, вы можете указать конкретный наполнитель, который должен выполняться, определив свойство `$seeder` в вашем тестовом классе:

```
use Database\Seeders\OrderStatusSeeder;

/**
 * Запускать указанный наполнитель перед каждым тестом.
 *
 * @var string
 */
protected $seeder = OrderStatusSeeder::class;
```

## # Доступные утверждения

Laravel содержит несколько утверждений базы данных для ваших функциональных тестов [Pest](#) или [PHPUnit](#). Мы обсудим каждое из этих утверждений ниже.

### assertDatabaseCount

Утверждает, что таблица в базе данных содержит указанное количество записей:

```
$this->assertDatabaseCount('users', 5);
```

### assertDatabaseEmpty

Утверждает, что таблица в базе данных не содержит записей:

```
$this->assertDatabaseEmpty('users');
```

### assertDatabaseHas

Утверждает, что таблица в базе данных содержит записи, соответствующие переданным ключ / значение ограничениям запроса:

```
$this->assertDatabaseHas('users', [  
    'email' => 'sally@example.com',  
]);
```

## assertDatabaseMissing

Утверждает, что таблица в базе данных не содержит записей, соответствующих переданным ключ / значение ограничениям запроса:

```
$this->assertDatabaseMissing('users', [  
    'email' => 'sally@example.com',  
]);
```

## assertSoftDeleted

Метод `assertSoftDeleted` используется для утверждения того, что переданная модель Eloquent была «программно удалена»:

```
$this->assertSoftDeleted($user);
```

## assertNotSoftDeleted

Метод `assertNotSoftDeleted` используется для утверждения того, что переданная модель Eloquent была «программно удалена»

```
$this->assertNotSoftDeleted($user);
```

## assertModelExists

Утверждает, что данная модель существует в базе данных:

```
use App\Models\User;  
  
$user = User::factory()->create();
```

```
$this->assertModelExists($user);
```

## assertModelMissing

Утверждает, что данной модели не существует в базе данных:

```
use App\Models\User;  
  
$user = User::factory()->create();  
  
$user->delete();  
  
$this->assertModelMissing($user);
```

## expectsDatabaseQueryCount

Метод `expectsDatabaseQueryCount` может быть вызван в начале вашего теста для указания общего числа запросов к базе данных, которые вы ожидаете во время выполнения теста. Если фактическое количество выполненных запросов не соответствует ожиданиям, тест завершится неудачей:

```
$this->expectsDatabaseQueryCount(5);  
  
// Test...
```

# Тестирование · Имитация (Мок)

- # Введение
- # Подставные объекты
- # Имитация фасадов
  - # Шпионы фасадов
- # Взаимодействие со временем

## # Введение

При тестировании приложений Laravel бывает необходимо «сымитировать» определенные аспекты вашего приложения, чтобы они фактически не выполнялись во время текущего теста. Например, при тестировании контроллера, который инициирует событие, вы можете смоделировать слушателей событий, чтобы они фактически не выполнялись во время теста. Это позволяет вам тестировать только HTTP-ответ контроллера, не беспокоясь о запуске слушателей событий, поскольку слушатели событий могут быть протестированы в их собственном тестовом классе.

Laravel предлагает полезные методы для имитации событий, заданий и других фасадов из коробки. Эти помощники в первую очередь обеспечивают удобную обертку над Mockery, поэтому вам не нужно вручную выполнять сложные вызовы методов Mockery.

## # Подставные объекты

При имитации объекта, который будет внедрен в ваше приложение через [контейнер служб](#) Laravel, вам нужно будет привязать ваш подставной экземпляр к контейнеру с помощью `instance`. Это даст указание контейнеру использовать ваш подставной экземпляр объекта вместо создания самого объекта:

Pest      PHPUnit

```
use App\Service;  
use Mockery;
```

```
use Mockery\MockInterface;

test('something can be mocked', function () {
    $this->instance(
        Service::class,
        Mockery::mock(Service::class, function (MockInterface $mock) {
            $mock->shouldReceive('process')->once();
        })
    );
});
```

Чтобы сделать это более удобным, вы можете использовать метод `mock`, который обеспечен базовым классом тестов Laravel. Например, следующий пример эквивалентен приведенному выше примеру:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->mock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

Вы можете использовать метод `partialMock`, когда вам нужно только имитировать несколько методов объекта. Методы, которые не являются сымитированными, при вызове будут выполняться в обычном режиме:

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->partialMock(Service::class, function (MockInterface $mock) {
    $mock->shouldReceive('process')->once();
});
```

Точно так же, если вы хотите [шпионить](#) за объектом, базовый класс тестов Laravel содержит метод `spy` в качестве удобной обертки для метода `Mockery::spy`. Шпионы похожи на подставные объекты; однако, шпионы записывают любое взаимодействие между шпионом и тестируемым кодом, позволяя вам делать утверждения после выполнения кода:

```
use App\Service;

$spy = $this->spy(Service::class);
```

```
// ...  
  
$spy->shouldReceive('process');
```

## # Имитация фасадов

В отличие от традиционных вызовов статических методов, [фасады](#), включая [фасады в реальном времени](#) можно имитировать. Это дает большое преимущество перед традиционными статическими методами и дает вам такую же возможность тестирования, как если бы вы использовали традиционное внедрение зависимостей. При тестировании вы часто можете имитировать вызов фасада Laravel, происходящий в одном из ваших контроллеров. Например, рассмотрим следующее действие контроллера:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Support\Facades\Cache;  
  
class UserController extends Controller  
{  
    /**  
     * Получить список всех пользователей приложения.  
     */  
    public function index(): array  
    {  
        $value = Cache::get('key');  
  
        return [  
            // ...  
        ];  
    }  
}
```

Мы можем имитировать вызов фасада [Cache](#), используя метод [shouldReceive](#), который вернет экземпляр [Mockery](#). Поскольку фасады фактически извлекаются и управляются [контейнером служб](#) Laravel, то они имеют гораздо большую testируемость, чем типичный статический класс. Например, давайте сымитируем наш вызов метода [get](#) фасада [Cache](#):

Pest      PHPUnit

```
<?php

use Illuminate\Support\Facades\Cache;

test('get index', function () {
    Cache::shouldReceive('get')
        ->once()
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/users');

    // ...
});
```

Вы не должны имитировать фасад `Request`. Вместо этого передайте требуемые данные в [методы тестирования HTTP](#), такие как `get` и `post`, при запуске вашего теста. Аналогично, вместо имитации фасада `Config`, вызовите метод `Config::set` в ваших тестах.

## Шпионы фасадов

Если вы хотите [шпионить](#) за фасадом, то вы можете вызвать метод `spy` на соответствующем фасаде. Шпионы похожи на подставные объекты; однако, шпионы записывают любое взаимодействие между шпионом и тестируемым кодом, позволяя вам делать утверждения после выполнения кода:

Pest      PHPUnit

```
<?php

use Illuminate\Support\Facades\Cache;

test('values are be stored in cache', function () {
    Cache::spy();

    $response = $this->get('/');
```

```
$response->assertStatus(200);

Cache::shouldReceive('put')->once()->with(['name' , 'Taylor' , 10]);
});
```

## # Взаимодействие со временем

При тестировании вам может иногда потребоваться изменить время, возвращаемое такими помощниками, как `now` или `Illuminate\Support\Carbon::now()`. К счастью, базовый класс тестирования функций Laravel включает помощников, которые позволяют вам управлять текущим временем:

Pest      PHPUnit

```
test('time can be manipulated', function () {
    // Travel into the future...
    $this->travel(5)->milliseconds();
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // Travel into the past...
    $this->travel(-5)->hours();

    // Travel to an explicit time...
    $this->travelTo(now()->subHours(6));

    // Return back to the present time...
    $this->travelBack();
});
```

Вы также можете предоставить замыкание для различных методов путешествия во времени. Замыкание будет вызвано с замороженным временем в указанное время. После выполнения замыкания время возобновится как обычно:

```
$this->travel(5)->days(function () {
    // Test something five days into the future...
});
```

```
$this->travelTo(now()->subDays(10), function () {
    // Test something during a given moment...
});
```

Метод `freezeTime` может быть использован для замораживания текущего времени. Аналогично, метод `freezeSecond` заморозит текущее время, но в начале текущей секунды:

```
use Illuminate\Support\Carbon;

// Freeze time and resume normal time after executing closure...
$this->freezeTime(function (Carbon $time) {
    // ...
});

// Freeze time at the current second and resume normal time after executing closure.
$this->freezeSecond(function (Carbon $time) {
    // ...
})
```



Как и ожидалось, все обсуждаемые выше методы в основном полезны для тестирования поведения приложения, зависящего от времени, такого как блокировка неактивных сообщений на форуме:

Pest      PHPUnit

```
use App\Models\Thread;

test('forum threads lock after one week of inactivity', function () {
    $thread = Thread::factory()->create();

    $this->travel(1)->week();

    expect($thread->isLockedByInactivity())->toBeTrue();
});
```