Владимир Дронов

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Laravel 9

Быстрая разработка веб-сайтов на PHP

Модели, контроллеры и шаблоны
Маршрутизация
Разграничение доступа
CAPTCHA
BBCode
Аутентификация через социальные сети
Обработка событий
Оповещения
Отложенные задания
Планировщик
Локализация веб-сайтов
Разработка веб-служб REST
Публикация сайта

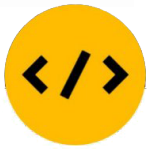
Материалы на www.bhv.ru

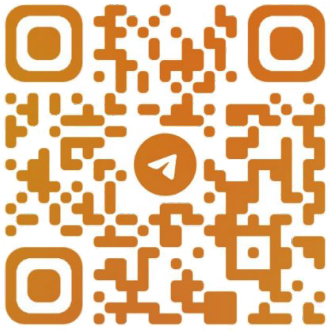
bhv®

Владимир Дронов

Laravel 9

Быстрая разработка веб-сайтов на PHP





Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.738.5+004.43
ББК 32.973.26-018.2
Д75

Дронов В. А.

Д75 Laravel 9. Быстрая разработка веб-сайтов на PHP. — СПб.: БХВ-Петербург, 2023. — 768 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-1725-6

Книга представляет собой полное описание фреймворка Laravel 9 для быстрой разработки сайтов на языке PHP. Дан краткий вводный курс для начинающих, в котором описывается разработка простого учебного сайта — электронной доски объявлений. Описаны базовые инструменты Laravel: миграции, модели, маршруты, контроллеры, шаблоны, средства обработки пользовательского ввода и сохранения выгруженных файлов, валидаторы, шаблоны, пагинаторы и инструменты разграничения доступа. Рассказано о более развитых средствах: внедрении зависимостей, провайдерах, посредниках, событиях и их обработке, отправке электронной почты, оповещениях, очередях и отложенных заданиях, встроенном планировщике, инструментах кеширования, локализации сайтов и расширении возможностей встроенной утилиты artisan. Описаны дополнительные библиотеки для обработки BBCode-тегов и CAPTCHA, вывода графических миниатюр, аутентификации через социальные сети. Рассмотрено программирование веб-служб REST, вещание по протоколу WebSocket и публикация сайта.

Электронный архив на сайте издательства содержит исходный код описанного в книге сайта.

Для веб-программистов

УДК 004.738.5+004.43
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-5-9775-1725-6

Оглавление

Предисловие

Laravel — сейчас самый популярный в мире PHP-фреймворк, предназначенный для разработки веб-сайтов. Более того, он остается номером один с 2015 года, до сих пор никому не уступив призовое место.

Почему именно Laravel?

Да потому, что:

- Laravel — полнофункциональный фреймворк.

  Он содержит все программные подсистемы, необходимые для разработки среднестатистического сайта: шаблонизатор, маршрутизатор, средства разграничения доступа, валидации, сохранения выгруженных файлов, базовую функциональность контроллеров и моделей. После установки самого фреймворка ничего доустанавливать не нужно.

- Богатейшими функциональными возможностями Laravel могут похвастаться далеко не все конкуренты.

  Хотите рассылать пользователям сайта короткие оповещения о каких-либо событиях (например, о появлении новых статей)? Нет ничего проще: подсистема оповещений Laravel позволяет отправлять такие оповещения не только традиционно, по электронной почте, но и по SMS. Хотите производить на сайте какие-либо технические работы по определенному расписанию (например, очищать мусорные данные каждый месяц)? Никаких проблем: встроенный планировщик Laravel запустит выполнение задачи в нужный момент. Может, желаете повысить отзывчивость сайта, перенеся наиболее «медленные» операции (скажем, рассылку почты) в параллельный процесс? И это не составит труда — достаточно лишь задействовать подсистему отложенных заданий!

- У Laravel низкий порог вхождения.

  Для программирования сайтов с применением Laravel достаточно базовых знаний PHP и основ веб-разработки. Фреймворк не требует сложного конфигурирования и готов к работе сразу после установки. Отдельные модули, составляющие код сайта, не требуется явно связывать друг с другом — достаточно «разложить» их по нужным папкам, и сайт будет прекрасно работать.

- Существует множество дополнительных библиотек, расширяющих функциональность фреймворка, и программ, помогающих в работе.

  Дополнительные библиотеки добавляют фреймворку поддержку BBCode, создания миниатюр графических изображений, выполнения входа на сайт через сторонние интернет-службы (например, социальные сети) и многое другое. В число дополнительных программ входят мощные отладочные панели, гибко настраиваемые административные подсистемы, различные инструментальные утилиты и пр. Часть этих библиотек и программ написана самим сообществом разработчиков Laravel.

На фоне столь значимых достоинств теряются отдельные недостатки фреймворка:

- местами — чрезмерная функциональность.

  Например, существуют функции `__()` (два подчеркивания) и `trans()`, выполняющие одинаковое действие. Ряд классов содержат методы, выполняющие одну и ту же задачу. Зачем это было сделано — непонятно…

- плохая официальная документация.

  Она неполна, местами поверхностна, местами хаотична. Некоторые ключевые моменты не описаны, и приходится искать информацию о них в сторонних источниках и даже в программном коде фреймворка. Полностью отсутствует руководство для начинающих, дающее основные знания в процессе разработки какого-либо учебного сайта.

> ***Внимание!***
>
> Автор предполагает, что читатели этой книги знакомы с языками HTML, CSS, JavaScript, PHP, принципами работы СУБД и имеют базовые навыки в веб-разработке. В книге все это описываться не будет.

О чем эта книга?

Автор книги поставил перед собой задачи:

- привести в начале книги вводный курс для начинающих.

  В нем описано программирование учебного веб-сайта — простой доски объявлений. И попутно объясняются основные понятия и принципы Laravel;

- дать максимально полное описание всех программных инструментов Laravel, применяемых при разработке среднестатистического сайта.

  За кадром остались лишь средства, чья полезность, по мнению автора, сомнительна (наподобие подсистемы автоматического тестирования), или применяемые в крайне специфических случаях (скажем, встроенный HTTP-клиент), а также создание дополнительных служб и библиотек для Laravel. Также не рассмотрены наиболее сложные дополнительные библиотеки и программы: средства для электронной коммерции, отладочные панели и административные подсистемы — поскольку объем книги ограничен;

- в том числе — рассказать о том, о чем молчит официальная документация.

  Для этого автор изучал сторонние источники информации, программный код Laravel и активно экспериментировал;

- привести побольше практических примеров применения того или иного программного инструмента.

  Код многих примеров взят с работающего экспериментального сайта, написанного автором.

> ***Электронный архив***
>
> Сопровождающий книгу электронный архив содержит программный код учебного сайта электронной доски объявлений, разработка которого описывается в *части I* книги. Архив доступен для скачивания с сервера издательства «БХВ» по ссылке **https://zip.bhv.ru/9785977517256.zip**, ссылка на него также ведет со страницы книги на сайте **https://bhv.ru/** (см. *приложение*).

Используемое ПО

Автор применял в работе над книгой следующее ПО:

- Microsoft Windows 10, русская 64-разрядная редакция со всеми установленными обновлениями;
- PHP — 8.1.5;
- Composer — 2.3.5;
- библиотека-комплект laravel/laravel — 9.1.1, включающая:
  - фреймворк Laravel (laravel/framework) — 9.9.0;
  - консоль Laravel (laravel/tinker) — 2.7.2;
  - Laravel Mix — 6.0.49;
- laravel/ui — 3.4.5;
- Laravel Lang — 10.7.1;
- LaravelLang — 10.0.0[1];
- doctrine/dbal — 3.3.6;
- Laravel HTML — 6.3.0;
- genert/bbcode — 1.1.2;
- Captcha for Laravel — 3.2.7;
- bkwld/croppa — 6.0.1;
- Laravel Socialite — 5.5.2;
- провайдер «ВКонтакте» для Laravel Socialite — 4.2.2;
- predis — 1.1.10;
- Laravel Websockets — 1.13.1;
- Laravel Echo — 1.12.0;
- pusher-js — 7.1.1-beta.

[1] Laravel Lang и LaravelLang — это разные библиотеки.

Типографские соглашения

В книге будут часто приводиться форматы написания различных языковых конструкций, применяемых в PHP. В них использованы особые типографские соглашения, приведенные далее.

- В угловые скобки (<>) заключаются наименования различных значений, подставляемых в исходный код (например, параметров функций и методов), которые дополнительно выделяются курсивом. Например:

  ```
  environment(<режим>)
  ```

  Здесь вместо слова *режим* должно быть подставлено реальное обозначение режима, в котором работает сайт.

- В квадратные скобки ([]) заключаются фрагменты кода, необязательные к указанию. Например:

  ```
  php artisan key:generate [--force]
  ```

  Здесь командный ключ `--force` может быть указан, а может и не указываться.

  У необязательных параметров функций и методов ставятся значения по умолчанию. Пример:

  ```
  unsignedInteger(<имя поля>[, <автоинкрементное?>=false])
  ```

  Здесь у необязательного параметра *автоинкрементное* значение по умолчанию `false`.

- Вертикальной чертой (|) разделяются различные варианты языковой конструкции, из которых следует указать лишь какой-то один. Пример:

  ```
  dropColumn(<имя поля>|<массив имен полей>)
  ```

  Здесь в качестве параметра метода `dropColumn()` следует поставить либо *имя поля*, либо *массив имен полей*.

- Слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывал на несколько строк и в местах разрывов ставил знак ↳. Например:

  ```
  php artisan queue:retry 49a473d6-198e-45df-a17e-4e885e7875fc ↳
  91401d2c-0784-4f43-824c-34f94a33c24d
  ```

  Приведенный код разбит на две строки, но должен быть набран в одну. Символ ↳ при этом вводить не нужно.

- Троеточием (. . .) помечены фрагменты кода, пропущенные ради сокращения объема текста. Пример:

  ```
  <table class="table table-striped table-borderless">
      . . .
  </table>
  ```

  Здесь пропущено содержимое тега `<table>`, не представляющее интереса при рассмотрении примера.

  Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные выражения, а оставшиеся неизмененны-

ми пропущены. Также троеточие используется, чтобы показать, в какое место должен быть вставлен вновь написанный код, — в начало исходного фрагмента, в его конец или в середину (между уже присутствующими в нем выражениями).

- Полужирным шрифтом выделен вновь добавленный или исправленный код. Пример:

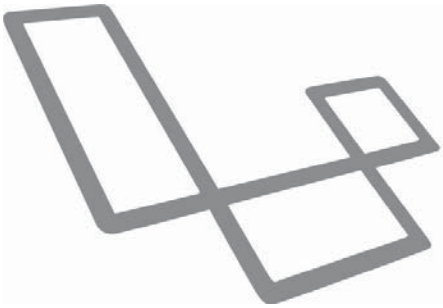
```
class Bb extends Model {
    . . .
    protected $fillable = ['title', 'content', 'price'];
}
```

Здесь в класс `Bb` был добавлен код, объявляющий свойство `fillable`.

- Зачеркнутым шрифтом выделяется код, подлежащий удалению. Пример:

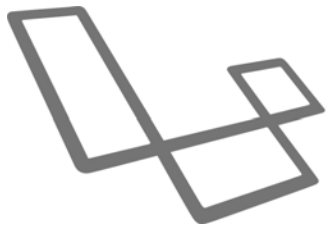
```
public function detail(Bb $bb) {
    ~~$bb = Bb::find($bb);~~
    $s = $bb->title . "\r\n\r\n";
    . . .
}
```

Первое выражение тела метода `detail()` следует удалить.



ЧАСТЬ I

Основы Laravel на практическом примере



ГЛАВА 1

Простейший веб-сайт — доска объявлений

В этой главе мы начнем писать с применением Laravel простой сайт — электронную доску объявлений. И на практическом примере рассмотрим принципы, положенные в основу этого PHP-фреймворка.

1.1. Подготовительные действия

1. Установим исполняющую среду PHP. Ее дистрибутив и инструкции по установке можно найти на «домашнем» сайте платформы (**https://www.php.net/**).

   Для запуска разрабатываемых сайтов удобнее всего применять отладочный веб-сервер, встроенный в PHP. Отдельную программу веб-сервера для этого использовать необязательно.

2. Установим программу Composer, чей дистрибутив находится на «домашнем» сайте этой утилиты (**https://getcomposer.org/**).

   Composer — это установщик PHP-библиотек и утилит вместе с зависимостями (библиотеками, используемыми устанавливаемой библиотекой). Composer самостоятельно ищет указанную библиотеку в интернет-репозитории **https://repo.packagist.org/**, загружает и распаковывает ее.

   В процессе установки Composer необходимо указать путь к файлу php.exe (консольной редакции исполняющей среды PHP) и предписать добавить этот путь в список путей системной переменной `PATH`, установив соответствующий флажок.

Установив необходимые программы, начнем разработку сайта, создав его проект.

1.2. Проект и его создание. Папка проекта

Теория

Проект — это совокупность файлов, хранящих программный, HTML- и CSS-код сайта, а также всевозможные дополнительные данные (например, параметры сайта и используемую им базу данных). Можно сказать, что проект — это и есть сайт.

Все файлы, составляющие проект, должны храниться в одной папке, называемой *папкой проекта*. Папка проекта может иметь произвольное местоположение в файловой системе.

При создании проекта его папка получает то же имя, что и сам проект. Однако в дальнейшем ее можно переименовать и даже переместить в другое место.

Во вновь созданном проекте следует установить дополнительную библиотеку laravel/ui, служащую для быстрого создания базовых средств разграничения доступа (с ними мы познакомимся в *главе 2*).

Практика

Создадим проект нашего сайта, дав ему имя `bboard`.

1. В командной строке выполним переход в папку, в которой будет находиться папка создаваемого проекта (у автора книги это папка C:\work\projects):

   ```
   c:
   cd \
   cd work\projects
   ```

2. Создадим проект `bboard`:

   ```
   composer create-project laravel/laravel bboard "9.1.*"
   ```

   Команда `create-project` программы Composer (установленной в *разд. 1.1*) создает новый проект PHP-сайта. После этой команды через пробелы последовательно указываются имя ключевой библиотеки, закладываемой в основу создаваемого проекта (у нас — `laravel/laravel`, то есть комплект из фреймворка Laravel и некоторых дополнительных библиотек), имя проекта (`bboard`) и версия ключевой библиотеки (`9.1.*`[1]).

   Через некоторое время в текущей папке появится папка bboard, содержащая только что созданный проект сайта.

3. Перейдем в папку проекта, отдав команду:

   ```
   cd bboard
   ```

4. Установим библиотеку laravel/ui командой:

   ```
   composer require laravel/ui
   ```

Папка проекта хранит множество вложенных папок и файлов. Так, папка app содержит все программные PHP-модули, составляющие код сайта, относящиеся к разным типам и «разложенные» по разным папкам, папка config — модули конфигурации, папка database\migrations — модули миграций, папка public является корневой папкой сайта (в ней можно сохранять файлы с таблицами стилей и веб-сценариями), папка resources\views хранит шаблоны, папка routes — модули со списками маршрутов, папка vendor — сам фреймворк и все его зависимости, а файл .env — локальные настройки проекта. Более подробно содержимое папки проекта мы рассмотрим в *главе 3*.

---

[1] Именно эта версия описывается в настоящей книге. В новых версиях могут использоваться другие версии входящих в состав комплекта библиотек и даже другие библиотеки, не описанные здесь.

Полезно знать

- Существует еще одна возможность создать новый Laravel-проект — посредством утилиты laravel (подробности — в *главе 3*). Однако с ее помощью невозможно указать версию Laravel, на которой будет основан создаваемый проект.
- Установщик Composer может устанавливать библиотеки и утилиты на уровне проекта или на уровне системы.
  - При установке на уровне проекта — библиотеки и утилиты записываются в папку *vendor* текущего проекта и доступны только в текущем проекте. Таким образом устанавливаются библиотеки, используемые в коде проекта. Установка на уровне проекта выполняется по умолчанию.
  - При установке на уровне системы — библиотеки и утилиты записываются в папке *<папка пользовательского профиля>\AppData\Roaming\Composer\vendor* и доступны везде. Таким образом устанавливаются инструментальные утилиты. Установка на уровне системы выполняется отдачей основной команды `global` и дополнительной команды `require`.

Только что созданный «пустой» проект Laravel можно запустить на исполнение и открыть в веб-обозревателе.

1.3. Запуск проекта. Отладочный веб-сервер PHP

1. В командной строке проверим, находимся ли мы в папке проекта (см. *разд. 1.2*), и если это не так, перейдем в эту папку.
2. Запустим отладочный веб-сервер PHP, набрав команду:

   ```
   php artisan serve
   ```

   Artisan — это утилита, написанная на PHP, хранящаяся непосредственно в папке проекта и служащая для выполнения различных действий над проектом. Команда `serve` этой утилиты запускает встроенный в PHP отладочный веб-сервер.
3. Прочитаем появившееся в командной строке сообщение, выведенное утилитой artisan:

   ```
   Starting Laravel development server: http://127.0.0.1:8000
   ```

   В нем говорится, что текущий проект Laravel-сайта скоро будет запущен и доступен через TCP-порт 8000.
4. Запустим веб-обозреватель и перейдем по интернет-адресу **http://localhost:8000/**.

   Веб-обозреватель выведет единственную страницу «пустого» сайта (рис. 1.1).

   В процессе работы отладочный веб-сервер будет выводить в командной строке журнал работы. В каждой строке журнала будут показываться: дата и время получения очередного клиентского запроса, интернет-адрес и TCP-порт, с которых пришел запрос, и статус серверного ответа.
5. Завершим работу отладочного веб-сервера, переключившись в командную строку, где он был запущен, и нажав комбинацию клавиш <Ctrl>+<Break> или <Ctrl>+<C>.

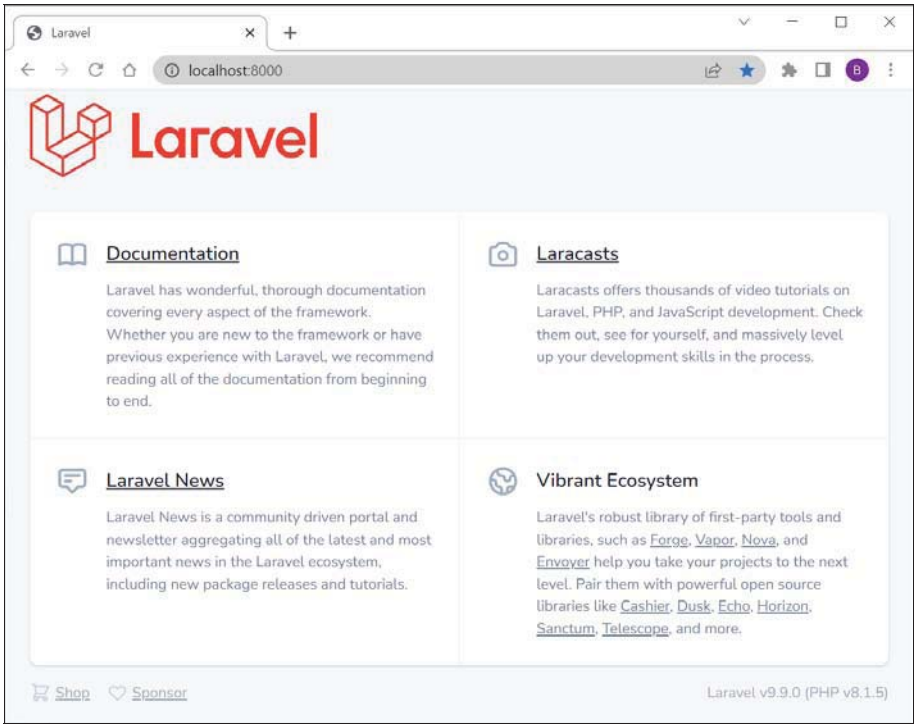


Рис. 1.1. Единственная веб-страница «пустого» веб-сайта Laravel

Если вы исправили исходный код, но не видите на экране результата правок...

...остановите отладочный веб-сервер и запустите его снова.

Имейте в виду, что отладочный сервер весьма медленный, и первый вывод страницы после значительных правок исходного кода может занять несколько секунд (далее та же страница будет выводиться быстрее).

Создадим теперь первый контроллер нашего сайта.

1.4. Контроллеры и действия

Теория

Контроллер — это программный модуль, реализующий функциональность одного из разделов сайта (например, раздела, выводящего объявления). *Действие* (action) — одна из операций, выполняемых контроллером (вывод страницы с перечнем объявлений, вывод отдельного объявления, вывод страницы для добавления объявления, сохранение добавленного объявления в базе и пр.).

Laravel позволяет создавать контроллеры трех разновидностей. Наиболее часто на практике применяются *контроллеры-классы*, реализуемые в виде классов. Контроллер-класс может содержать произвольное количество действий, каждое из которых реализуется в виде общедоступного метода.

По принятому в Laravel соглашению модули с контроллерами-классами сохраняются в папке app\Http\Controllers папки проекта, а имена их классов должны заканчиваться словом Controller.

Практика

Напишем контроллер-класс BbsController, обрабатывающий список объявлений, с действием index(), которое в будущем станет выводить страницу с перечнем объявлений, а сейчас — временную «заглушку» в виде обычного текста.

1. В командной строке проверим, находимся ли мы в папке проекта, и если это не так, перейдем в эту папку.

   Имейте в виду!
   В дальнейшем автор больше не будет напоминать об этом.

2. Создадим контроллер BbsController:

   ```
   php artisan make:controller BbsController
   ```

   Команда make:controller утилиты artisan создает модуль с классом контроллера, чье имя указано после команды через пробел.

3. Откроем только что сгенерированный модуль app\Http\Controllers\BbsController.php в текстовом редакторе и посмотрим на его код, приведенный в листинге 1.1 (служебный код и комментарии опущены ради краткости).

   Листинг 1.1. Код «пустого» контроллера BbsController

   ```
   namespace App\Http\Controllers;

   use Illuminate\Http\Request;

   class BbsController extends Controller {
   }
   ```

   Контроллер объявлен в пространстве имен App\Http\Controllers и является производным от класса Controller. Изначально он «пуст» — не содержит ни одного метода-действия.

4. Объявим в контроллере-классе BbsController действие index(), выводящее временную текстовую «заглушку»:

   ```
   class BbsController extends Controller {
       public function index() {
           return response('Здесь будет перечень объявлений.')
                  ->header('Content-Type', 'text/plain');
       }
   }
   ```

   Функция response() генерирует серверный ответ, представленный объектом класса Illuminate\Http\Response, на основе строки, переданной в параметре, и возвращает его в качестве результата.

Метод `header()` класса `Illuminate\Http\Response` помещает в текущий серверный ответ заголовок с заданными в параметрах именем и значением. Мы используем этот метод, чтобы поместить в ответ заголовок `Content-Type` со значением `text/plain`, тем самым указав веб-обозревателю, что ответ содержит обычный текст.

Готовый ответ следует вернуть из метода-действия в качестве результата — чтобы Laravel смог отправить его клиенту.

НЕ ЗАБЫВАЕМ СОХРАНЯТЬ ИСПРАВЛЕННЫЕ ФАЙЛЫ!

Автор далее не будет напоминать об этом.

5. Запустим отладочный веб-сервер и откроем разрабатываемый сайт в веб-обозревателе.

В результате мы опять увидим страницу, показанную на рис. 1.1, но не заданную нами текстовую «заглушку». А все потому, что мы не исправили маршрут.

СОВЕТ

Пользователи текстового редактора Visual Studio Code могут установить весьма полезный набор расширений Laravel Extension Pack. Он обеспечивает улучшенную синтаксическую подсветку, автозавершение кода, автоматическое создание шаблонов и многое другое.

ПОЛЕЗНО ЗНАТЬ

- В Laravel используется принятое в PHP соглашение, согласно которому вложенные друг в друга пространства имен, в которых объявлен класс, должны соответствовать вложенным друг в друга папкам файловой системы, в которых хранится модуль с кодом этого класса. Так, код класса `App\Http\Controllers\BbsController` должен храниться в модуле *app\Http\Controllers\BbsController.php*.

  Следование этому соглашению позволяет программному ядру Laravel быстро найти модуль с нужным классом.

- Класс `App\Http\Controllers\Controller`, являющийся базовым для всех контроллеров-классов Laravel, хранится в модуле *app\Http\Controllers\Controller.php*. Изначально он практически «пуст» — лишь включает три трейта (которые мы рассмотрим в следующих главах).

1.5. Маршруты и списки маршрутов. Фасады

Теория

Каждая операция, производимая сайтом (вывод страницы, сохранение введенных данных в базе и пр.), выполняется при получении им от веб-обозревателя клиентского запроса, выполненного по определенному пути с применением определенного HTTP-метода (GET, POST, PATCH и др.).

Путь — это часть интернет-адреса, находящаяся между адресом хоста и набором GET-параметров и идентифицирующая запрашиваемую страницу (например, интернет-адрес **http://localhost:8000/items/34?from=index** содержит путь **items/34**).

Следовательно, чтобы какое-либо действие контроллера выполнилось при получении запроса, выполненного по определенному пути определенным HTTP-методом, это действие следует связать с нужными путем и методом, создав *маршрут*.

Маршрут Laravel — это объект особого класса, содержащий следующие сведения:

- *шаблонный путь* — задает нужный формат путей;
- *допустимый HTTP-метод* — которым должен быть выполнен клиентский запрос;
- действие контроллера — выполняется при совпадении шаблонного пути и допустимого метода с путем и методом, извлеченными из полученного клиентского запроса (т. е. если маршрут является *совпавшим*).

В качестве примера рассмотрим следующие маршруты (записаны в формате «шаблонный путь — допустимый метод — выполняемая операция»):

- / (прямой слеш — «корень» сайта) — GET — вывод перечня объявлений;
- ***/<ключ объявления>/*** — GET — вывод объявления с заданным *ключом*;
- **/add/ —** GET — вывод страницы для добавления объявления;
- / — POST — сохранение добавленного объявления в базе.

Созданные маршруты записываются в особый *список*. Список *веб-маршрутов* (ведущих на действия контроллеров, которые выдают обычные страницы) хранится в модуле routes\web.php.

Просмотр списка маршрутов в поисках совпавшего выполняет подсистема фреймворка, называемая *маршрутизатором*. Если ни один маршрут не совпал, маршрутизатор выводит страницу с сообщением об ошибке 404 (запрашиваемый путь не поддерживается).

Практика

Создадим веб-маршрут, связывающий шаблонный путь / («корень» сайта) и допустимый HTTP-метод GET с действием `index()` контроллера `BbsController`.

1. Откроем модуль routes\web.php со списком веб-маршрутов в текстовом редакторе и посмотрим на имеющийся там код (листинг 1.2) — служебный код и комментарии в нем опущены.

Листинг 1.2. Код списка маршрутов routes\web.php

```
use Illuminate\Support\Facades\Route;

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

Класс `Route` — это фасад маршрутизатора (*фасадом* называется класс, служащий своего рода «пультом управления» одной из подсистем фреймворка). Статический метод `get()`, вызванный у этого фасада, указывает маршрутизатору создать новый объект маршрута, связывающий допустимый HTTP-метод GET (одноименный методу фасада), шаблонный путь из первого параметра (у нас — /, «корень» сайта) и контроллер, заданный вторым параметром. Здесь использован *контроллер-функция*, реализованный в виде функции и содержащий лишь одно действие. Он генерирует на основе шаблона welcome.blade.php (шаблонами мы займемся далее в этой главе) страницу, показанную на рис. 1.1.

2. Свяжем изначально созданный маршрут с действием `index()` контроллера `BbsController`, переписав его код следующим образом:

```
use App\Http\Controllers\BbsController;
Route::get('/', function () {
    return view('welcome');
});
Route::get('/', [BbsController::class, 'index']);
```

Запустим отладочный веб-сервер, откроем сайт и посмотрим на выведенную текстовую «заглушку» (рис. 1.2).

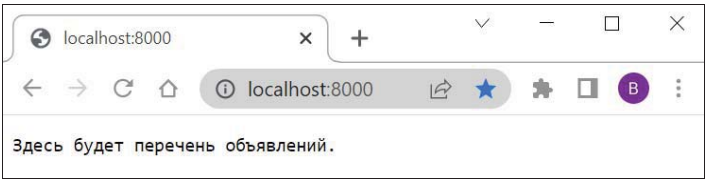


Рис. 1.2. Текстовая «заглушка», временно выводимая вместо перечня объявлений

Это была разминка. Далее мы займемся серьезным делом — создадим базу данных, запишем в нее пару-тройку объявлений и будем генерировать перечень объявлений на основе содержимого базы. Но сначала произведем необходимые настройки.

1.6. Настройки проекта. Подготовка проекта к работе с базой данных SQLite

Теория

Настройки Laravel-проекта делятся на два вида:

- *локальные* (или настройки *окружения*) — задают параметры, относящиеся к текущей платформе (в частности, имя базы данных и сведения для подключения к ней). *Не* заносятся в коммиты Git. Хранятся в файле .env, который располагается непосредственно в папке проекта;
- *рабочие* — задают все параметры разрабатываемого сайта. Заносятся в коммиты Git. Хранятся в папке config в виде 15 PHP-модулей, каждый из которых содержит настройки какой-либо из подсистем фреймворка. Настройки, записанные в таком модуле, организованы в виде ассоциативного массива PHP.

Практика

Настроим проект для работы с базой данных формата SQLite, хранящейся в файле database\data.sqlite, и заодно создадим эту базу данных.

1. Откроем файл .env, хранящий локальные настройки, в текстовом редакторе и найдем в нем фрагмент кода, настраивающий подключение к базе данных:

```
DB_CONNECTION=mysql
. . .
DB_DATABASE=laravel
```

2. Укажем формат базы данных SQLite и дадим файлу, хранящему базу, имя `data.sqlite`, исправив значения настроек `DB_CONNECTION` и `DB_DATABASE` следующим образом:

```
DB_CONNECTION=sqlite
. . .
DB_DATABASE=data.sqlite
```

   В настройке `DB_DATABASE` мы указали лишь имя файла, в то время как фреймворку для работы с базой SQLite требуется дать абсолютный путь к файлу. Сейчас сделаем так, чтобы этот путь формировался автоматически.

3. Откроем модуль config\database.php, хранящий рабочие настройки баз данных, в текстовом редакторе и найдем в нем следующий код:

```
return [
    . . .
    'connections' => [
        'sqlite' => [
            . . .
            'database' => env('DB_DATABASE',
                              database_path('database.sqlite')),
            . . .
        ],
        . . .
    ],
    . . .
];
```

   Настройка `connections.sqlite.database` задает абсолютный путь к файлу базы данных SQLite.

   Функция `env()` возвращает значение, записанное в файле .env, в настройке, чье имя указано в первом параметре. Если в файле .env нет такой настройки, функция `env()` возвращает значение заданного в ней второго параметра.

   Функция `database_path()` принимает в качестве параметра относительный путь к файлу, заданный от папки database, и возвращает абсолютный путь этого файла.

   Таким образом, изначально в рабочую настройку `connections.sqlite.database` заносится значение локальной настройки `DB_DATABASE` или, если таковая отсутствует, — абсолютный путь к несуществующему файлу database\database.sqlite.

   К сожалению, значение, извлекаемое из настройки `DB_DATABASE` файла .env, не «пропускается» через функцию `database_path()`, что вынуждает программистов указывать там абсолютный путь к файлу с базой. Устраним эту досадную недоработку разработчиков Laravel, для чего…

4. …перепишем приведенный ранее код следующим образом:

```
. . .
            'database' => database_path(env('DB_DATABASE',
                                            'database.sqlite')),
. . .
```

5. Создадим в папке database «пустой» файл data.sqlite.

   Проще всего сделать это, воспользовавшись контекстным меню **Создать** | **Текстовый документ** и переименовав получившийся файл в data.sqlite.

Итак, база данных у нас есть. Осталось создать в ней необходимые таблицы, поля, индексы и связи. Используем для этого миграцию.

> ***Полезно знать***
>
> Функции `env()`, `database_path()`, `response()` (рассмотрена в *разд. 1.4*) и многие другие, описанные в следующих главах, могут быть вызваны в любом месте кода проекта и носят название *хелперов*.

1.7. Миграции

Теория

Миграция — программный PHP-модуль, вносящий какие-либо изменения в структуру базы данных. Миграция может, например, создать таблицу вместе со всеми полями и индексами, исправить имя или тип поля в существующей таблице, создать или удалить индекс. Миграция реализуется в виде класса.

Миграцию можно применить или откатить. В процессе *применения* миграция выполняет все описанные в ней действия. При *откате* же она возвращает базу данных в состояние, существовавшее перед применением этой миграции. Понятно, что откатить можно лишь миграцию, примененную ранее.

Программные модули с миграциями хранятся в папке database\migrations. Список всех примененных миграций в хронологическом порядке сохраняется в особой таблице базы данных, автоматически создаваемой перед применением самой первой миграции.

Практика

Напишем миграцию `create_bbs_table`, создающую в базе данных таблицу `bbs` со следующими полями:

- `title` — заголовок объявления с названием продаваемого товара (тип — строковый, длина — 50 символов);
- `content` — сам текст объявления, описание товара (тип — memo);
- `price` — цена (тип — вещественное число).

1. В командной строке создадим «пустую» миграцию `create_bbs_table`:

   ```
   php artisan make:migration create_bbs_table --create=bbs
   ```

   Команда `make:migration` утилиты artisan создает миграцию с заданным именем. Дополнительный параметр `--create` предписывает вставить в миграцию код, создающий таблицу, чье имя указано в параметре.

2. Откроем только что созданный модуль с именем формата database\migrations\ *<текущая временна́я отметка>*_create_bbs_table.php, хранящий созданную нами ми-

грацию, в текстовом редакторе и посмотрим на содержащийся в нем код (листинг 1.3).

Листинг 1.3. Код «пустой» миграции

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

return new class extends Migration {
    public function up() {
        Schema::create('bbs', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

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

Этот код создает и возвращает объект анонимного класса, который и представляет миграцию. Класс миграции содержит два метода: `up()`, выполняющийся в процессе применения миграции, и `down()`, запускаемый при ее откате. Параметр `--create` предписывает утилите artisan сразу же вставить в эти методы код, соответственно создающий и удаляющий заданную в параметре таблицу.

В методе `up()` миграции выполняется создание таблицы. Для этого у фасада `Schema`, за которым «прячется» подсистема, манипулирующая структурой базы данных, вызывается метод `create()`. В первом параметре методу передается имя создаваемой таблицы, а во втором — анонимная функция, в качестве параметра принимающая объект класса `Blueprint`, который представляет структуру создаваемой таблицы. Добавление полей в нее выполняется вызовом различных методов у этого объекта.

Параметр `--create` также предписывает утилите artisan вставить в код этой функции вызовы методов `id()` и `timestamps()`. Первый метод добавляет в формируемую таблицу поле `id`, хранящее *ключ* (какое-либо уникальное значение, однозначно идентифицирующее запись, обычно — автоматически генерируемый уникальный номер). Поле, хранящее ключ, называется *ключевым*). Второй метод добавляет поля для хранения временны́х отметок создания записи (*отметка создания*) и ее последней правки (*отметка правки*).

В методе `down()` производится удаление таблицы. Для этого у фасада `Schema` вызывается метод `dropIfExists()`, которому передается имя удаляемой таблицы.

3. Добавим в метод `up()` миграции код, создающий поля `title`, `content` и `price`:

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

```
            $table->string('title', 50);
            $table->text('content');
            $table->float('price');
            $table->timestamps();
        });
    }
```

Метод `string()` класса `Blueprint` создает строковое поле типа `VARCHAR` с именем и длиной, заданными в параметрах метода. Для создания поля memo типа `TEXT` и поля вещественного типа `FLOAT` используются методы `text()` и `float()`.

Мы будем выводить объявления в хронологическом порядке, отсортированными по значению поля отметки создания записи. Это поле создается вызовом метода `timestamps()` и по умолчанию имеет имя `created_at`. Для ускорения сортировки создадим по нему индекс.

4. Добавим код, создающий обычный индекс по полю `created_at`:

```
public function up() {
    Schema::create('bbs', function (Blueprint $table) {
        . . .
        $table->timestamps();
        $table->index('created_at');
    });
}
```

Мы вызвали метод `index()` класса `Blueprint`, задав в параметре имя индексируемого поля.

5. Применим только что созданную миграцию, набрав в командной строке команду:

```
php artisan migrate
```

В результате Laravel выполнит все еще не выполненные миграции, находящиеся в папке database\migrations.

> ***ПОЛЕЗНО ЗНАТЬ***
>
> Только что созданный проект Laravel изначально содержит четыре миграции, создающие служебные таблицы, и, в частности, таблицы со списками зарегистрированных пользователей (подробности будут приведены в *главе 13*) и проваленных заданий (см. *главу 25*). При первом выполнении миграций они также будут выполнены.

1.8. Модели

Модель — программный модуль, служащий для взаимодействия с обслуживаемой им таблицей базы данных: выборки записей, извлечения значений полей, добавления, правки и удаления записей. Модель реализуется в виде класса.

PHP-модули с моделями по умолчанию хранятся в папке app\Models.

Напишем модель `Bb`, предназначенную для работы с таблицей `bbs` базы данных.

1. В командной строке создадим модуль с классом модели:

```
php artisan make:model Bb
```

2. Откроем модуль app\Models\Bb.php, хранящий созданную модель, в текстовом редакторе и посмотрим на его содержимое (листинг 1.4).

Листинг 1.4. Код «пустой» модели Bb

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Bb extends Model {
    use HasFactory;
}
```

Всю функциональность модели, необходимую для работы с базой данных, реализует базовый класс `Illuminate\Database\Eloquent\Model`. Трейт `Illuminate\Database\Eloquent\Factories\HasFactory` используется лишь при автоматизированном тестировании (которое в этой книге не описывается), так что его можно удалить.

3. Сделаем поля `title`, `content` и `price` доступными для массового присваивания (о нем — чуть позже), добавив в класс модели выделенный полужирным шрифтом код, и заодно удалим трейт `HasFactory`:

```
~~use Illuminate\Database\Eloquent\Factories\HasFactory;~~

class Bb extends Model {
    ~~use HasFactory;~~
    protected $fillable = ['title', 'content', 'price'];
}
```

Массив с именами полей, доступных для массового присваивания, заносится в защищенное свойство `fillable`.

1.9. Консоль Laravel

Консоль Laravel позволяет работать с классами фреймворка в интерактивном режиме. В частности, с ее помощью удобно заносить в информационную базу какие-либо отладочные данные.

1. В командной строке запустим консоль Laravel:

```
php artisan tinker
```

Приглашение к вводу команды обозначается префиксом `>>>`. Вывод результатов производится без всякого префикса.

2. Проверим консоль в работе, выведя полное имя класса модели `Bb`, для чего наберем следующие выражения, завершая каждое из них нажатием клавиши <Enter>:

```
>>> use App\Models\Bb;
>>> echo Bb::class;
App\Models\Bb
```

Следует помнить, что префиксом >>> обозначается приглашение к вводу команды. Результат выполнения команды выводится без префикса.

Консоль Laravel позволяет набирать выражения в несколько строк, разрывая их нажатием клавиши <Enter>.

3. Введем последнее из набранных выражений в две строки:

```
>>> use
... App\Models\Bb;
```

Префиксом ... (три точки) обозначается приглашение к вводу следующей строки многострочного выражения.

4. Завершим работу консоли Laravel, нажав комбинацию клавиш <Ctrl>+<Break> или <Ctrl>+<C>.

Также можно набрать в консоли команду `exit`.

1.10. Работа с базой данных

Добавим несколько записей в таблицу объявлений `bbs`, исправим какую-либо запись, удалим другую и попробуем выполнить выборку записей с фильтрацией и сортировкой. Затем переделаем действие `index()` контроллера `HomeController` таким образом, чтобы оно выводило перечень объявлений из базы данных.

1. Запустим консоль Laravel (как это сделать, было рассказано в *разд. 1.9*).
2. Добавим первое объявление, набрав код:

```
>>> use App\Models\Bb;
>>> $bb = new Bb();
=> App\Models\Bb {#3530}
>>> $bb->title = 'Шкаф';
=> "Шкаф"
>>> $bb->content = 'Совсем новый, полированный, двухстворчатый';
=> "Совсем новый, полированный, двухстворчатый"
>>> $bb->price = 2000;
=> 2000
>>> $bb->save();
=> true
```

Если выполненное выражение возвращает какой-либо результат, но не предполагает его явного вывода, результат все равно выводится в следующей строке, предваренный префиксом =>.

> ***Внимание!***
>
> Далее результаты, выводящиеся в консоли неявно и не нужные для понимания изложенного в книге материала, для краткости показываться не будут.

Объект модели представляет одну запись таблицы. Следовательно, для добавления записи мы можем создать новый объект модели, занести в его свойства, представляющие отдельные поля, нужные значения и вызвать метод `save()`, выполняющий сохранение записи. Что мы и сделали.

Метод `save()` вернет значение `true` (см. приведенный ранее код) — это значит, что запись была успешно сохранена.

3. Удостоверимся, что объявление действительно сохранилось в таблице, для чего выведем его ключ, хранящийся в поле `id`:

```
>>> echo $bb->id;
1
```

Как видим, запись № 1 действительно сохранилась.

4. Добавим другое объявление — другим способом:

```
>>> $bb = $bb->create(['title' => 'Пылесос',
...     'content' => 'Старый, ржавый, без шланга', 'price' => 1000]);
=> App\Models\Bb {#4476
     title: "Пылесос",
     content: "Старый, ржавый, без шланга",
     price: 1000,
     updated_at: "2022-04-25 11:27:25",
     created_at: "2022-04-25 11:27:25",
     id: 2,
   }
```

Отметим, что содержимое созданного этим способом объекта записи будет выведено непосредственно в консоли, и мы сразу сможем проверить правильность ввода данных.

Метод `create()` создает новую запись, заносит в ее поля значения из указанного ассоциативного массива, сохраняет запись и возвращает ее в качестве результата. Он использует *массовое присваивание*, при котором значения заносятся сразу в несколько полей создаваемой записи. Отметим, что таким образом можно занести значения только в поля, приведенные в списке доступных для массового присваивания (задается свойством `fillable` класса модели — см. *разд. 1.8*).

Любопытно, что метод `create()`, хотя и вызывается у модели, но выполняется *построителем запросов* — подсистемой, которая на основании заданных нами параметров (значений, заносимых в создаваемую запись, условий фильтрации выбираемых записей, их сортировки, набора выводимых полей и др.) формирует готовый SQL-запрос, отправляет его СУБД, получает результат и представляет его в удобном для обработки виде. При попытке вызвать метод построителя запросов у модели ее объект создает новый объект построителя запросов и «передает» ему вызов метода.

5. Добавим еще два объявления:

```
>>> $bb = Bb::create(['title' => 'Грузовик',
...     'content' => 'Грузоподъемность - 5 т', 'price' => 10000000]);
>>> $bb = Bb::create(['title' => 'Снег', 'content' => 'Прошлогодний',
...                                     'price' => 50]);
```

Методы построителя запроса можно вызывать не только у объекта модели, но и у ее класса — как статические.

6. Извлечем запись № 2:

```
>>> $bb = Bb::find(2);
```

Метод `find()`, опять же выполняемый построителем запросов, ищет и возвращает объект модели, хранящий запись с заданным ключом.

7. Посмотрим, что это за объявление:

```
>>> echo $bb->title, ' | ', $bb->content, ' | ', $bb->price;
Пылесос | Старый, ржавый, без шланга | 1000.0
```

Получить значения, хранящиеся в полях записи, можно, обратившись к одноименным свойствам модели.

Тысяча рублей за старый пылесос без шланга — не многовато ли?.. Уценим его.

8. Изменим цену товара (значение поля `price`):

```
>>> $bb->price = 500;
>>> $bb->save();
```

Мы занесли новое значение в нужное свойство модели и вызвали у объекта записи метод `save()`, чтобы сохранить исправленную запись.

9. Извлечем все объявления, отсортированные по увеличению цены:

```
>>> $bbs = Bb::orderBy('price')->get();
```

Метод `orderBy()` построителя запросов сортирует записи по указанному полю. В качестве результата он возвращает тот же объект построителя запросов, что позволяет записывать «цепочки» методов.

Мы так и сделали, «сцепив» с методом `orderBy()` метод `get()`, который отправит базе данных готовый SQL-запрос и вернет результат его выполнения в виде коллекции объектов модели `Bb`, хранящих выбранные записи.

10. Просмотрим отсортированные объявления:

```
>>> foreach ($bbs as $bb) {
...   echo $bb->title, ' | ', $bb->content, ' | ', $bb->price, "\r\n";
... }
Снег | Прошлогодний | 50.0
Пылесос | Старый, ржавый, без шланга | 500.0
Шкаф | Совсем новый, полированный, двухстворчатый | 2000.0
Грузовик | Грузоподъемность - 5 т | 10000000.0
```

Коллекцию записей, возвращенную методом `get()`, как и любую другую, можно перебрать в цикле.

11. Извлечем объявления с ценами более 1000 рублей и выведем их в обратном хронологическом порядке:

```
>>> $bbs = Bb::where('price', '>', 1000)->latest()->get();
>>> foreach ($bbs as $bb) {
...     echo $bb->title, ' | ', $bb->created_at, "\r\n";
... }
Грузовик | 2020-04-13 12:11:59
Шкаф | 2020-04-13 10:27:12
```

Метод `where()` построителя запросов фильтрует записи по значению поля, указанного в первом параметре. Второй параметр задает SQL-оператор сравнения, а третий — сравниваемое значение.

Метод `latest()` сортирует записи по убыванию отметок их создания (т. е. в обратном хронологическом порядке). Знакомый нам метод `get()` выполнит запрос и вернет результат.

12. Удалим объявление о продаже прошлогоднего снега (вряд ли на него будет спрос):

```
>>> $bb = Bb::where('title', 'Снег')->first();
>>> $bb->delete();
```

Ищем объявление вызовом метода `where()` (поскольку оператор сравнения в нем не указан, будет использован оператор равенства `=`). Метод `first()` возвращает первую запись из полученного результата. А метод `delete()` модели удаляет текущую запись.

13. Добавим еще несколько объявлений о продаже каких-либо товаров с произвольными содержанием и ценами.

14. Откроем модуль app\Http\Controllers\BbsController.php, хранящий код контроллера `BbsController`, в текстовом редакторе и переделаем действие `index()`, чтобы оно выводило перечень объявлений, взятых из базы данных, в обратном хронологическом порядке:

```
use App\Models\Bb;
class BbsController extends Controller {
    public function index() {
        $bbs = Bb::latest()->get();
        $s = "Объявления\r\n\r\n";
        foreach ($bbs as $bb) {
            $s .= $bb->title . "\r\n";
            $s .= $bb->price . " руб.\r\n";
            $s .= "\r\n";
        }
        return response('Здесь будет перечень объявлений.'$s)
             ->header('Content-Type', 'text/plain');
    }
}
```

Запустив отладочный веб-сервер и открыв сайт, мы увидим перечень, показанный на рис. 1.3.

Теперь сделаем так, чтобы при переходе по пути формата /*<ключ объявления>*/ наш сайт выводил объявление с записанным в пути *ключом*.

1.11. URL-параметры. Внедрение зависимостей

Теория

URL-параметр — значение, присутствующее в пути, который был получен в составе клиентского запроса, и предназначенное для программной обработки. Обозначение URL-параметра записывается в шаблонном пути под уникальным именем.

Значение, передаваемое в URL-параметре, впоследствии отправляется связанному с маршрутом действию контроллера через параметр этого действия с именем, совпадающим с именем самого URL-параметра.

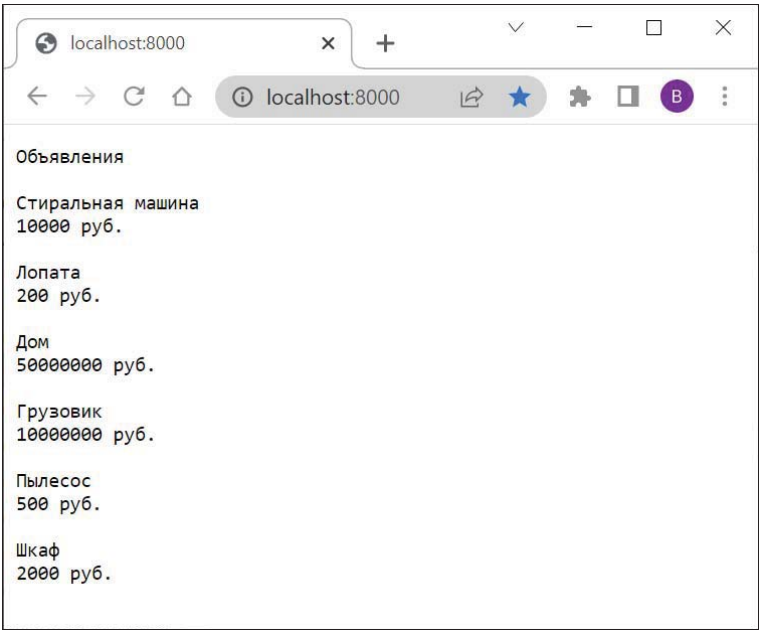


Рис. 1.3. Перечень объявлений, извлеченный из базы данных

Например, чтобы извлечь из пути формата ***/<ключ объявления>/*** записанный там *ключ*, мы можем создать в шаблонном пути маршрута URL-параметр с именем `bb`. Чтобы отправить его действию, связанному с этим маршрутом, мы объявим в действии параметр с тем же именем — `bb`.

Совпадение имен URL-параметра и параметра действия служит фреймворку своего рода «подсказкой» занести в параметр действия значение одноименного URL-параметра. Такого рода автоматическое занесение в параметры методов требуемых значений, основанное на совпадении имен и (или) типов этих параметров, называется *внедрением зависимостей*.

Практика

Реализуем вывод отдельного объявления, выбранного посетителем сайта. Для этого добавим в контроллер `BbsController` действие `detail()`, которое будет выполняться при получении запроса, произведенного HTTP-методом GET по пути формата ***/<ключ объявления>/***. Для извлечения из пути *ключа* объявим в шаблонном пути URL-параметр `bb`.

1. Откроем модуль routes\web.php, хранящий список веб-маршрутов, и добавим в него маршрут, связанный с действием `detail()` контроллера `HomeController`:

```
Route::get('/', [BbsController::class, 'index']);
Route::get('/{bb}', [BbsController::class, 'detail']);
```

   Конечный слеш в шаблонных путях не ставится. Обозначения URL-параметров берутся в фигурные скобки (например, `{bb}` — это URL-параметр с именем `bb`).

2. Откроем модуль app\Http\Controllers\BbsController.php, хранящий код контроллера BbsController, в текстовом редакторе и добавим действие detail():

```
class BbsController extends Controller {
    . . .
    public function detail($bb) {
        $bb = Bb::find($bb);
        $s = $bb->title . "\r\n\r\n";
        $s .= $bb->content . "\r\n";
        $s .= $bb->price . " руб.\r\n";
        return response($s)->header('Content-Type', 'text/plain');
    }
}
```

В объявлении метода detail() мы указали параметр с именем bb. Laravel сразу «сообразит», что в него нужно поместить значение одноименного URL-параметра.

3. Запустим отладочный сервер и выполним переход на объявление № 1, набрав интернет-адрес **http://localhost:8000/1/**. Результат показан на рис. 1.4.

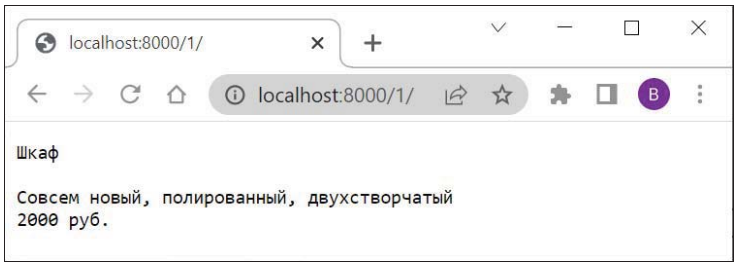


Рис. 1.4. Объявление № 1

Попробуем вывести объявления с номерами 2, 3 и т. д., но только не объявление № 4, которое мы удалили (если попытаемся вывести его, получим страницу с сообщением об ошибке 404).

Теперь немного сократим код действия detail(), предписав фреймворку помещать в параметр bb действия detail() не ключ выводимой записи, а сразу объект этой записи.

4. Внесем в код действия detail() контроллера BbsController следующую правку:

```
public function detail(Bb $bb) {
    $bb = Bb::find($bb);
    $s = $bb->title . "\r\n\r\n";
    . . .
}
```

Мы указали у параметра bb действия класс модели Bb в качестве типа — тогда «сообразительный» Laravel сам найдет запись по полученному ключу и подставит содержащий ее объект модели в этот параметр. Выражение, выполняющее поиск записи, станет ненужным и может быть удалено.

Снова протестируем сайт и убедимся, что он работает.

Полезно знать

Не все веб-фреймворки реализуют внедрение зависимостей. Например, популярнейший фреймворк Django, написанный на языке Python, такого не умеет.

Настала пора сделать так, чтобы наш сайт выдавал результаты не обычным текстом, а в виде полноценных веб-страниц.

1.12. Шаблоны

Теория

Шаблон — это образец для генерирования веб-страницы, отправляемой клиенту в составе ответа. Процесс генерирования страницы называется *рендерингом*, а подсистема фреймворка, выполняющая рендеринг, — *шаблонизатором*.

Для рендеринга шаблонизатору, помимо шаблона, нужны данные, которые будут выводиться на генерируемой странице (в нашем случае это, в зависимости от страницы, перечень объявлений или содержание выбранного объявления). Эти данные оформляются в виде особого набора, называемого *контекстом шаблона*. Контекст шаблона формируется в виде ассоциативного массива, который преобразуется шаблонизатором в набор обычных переменных, доступных внутри шаблона.

В Laravel шаблоны сохраняются в файлах с расширением blade.php и помещаются в папке resources\views или вложенных в нее папках.

Практика

Реализуем вывод перечня объявлений и содержимого выбранного объявления в виде обычных веб-страниц, для чего напишем шаблоны index.blade.php и detail.blade.php соответственно. Для оформления страниц применим популярный CSS-фреймворк Bootstrap наиболее актуальной на момент подготовки книги версии 5.1 (**https://getbootstrap.com/**).

1. Откроем модуль app\Http\Controllers\BbsController.php, хранящий код контроллера `BbsController`, и перепишем действие `index()` таким образом, чтобы оно формировало страницу на основе шаблона index.blade.php:

```
public function index() {
    $context = ['bbs' => Bb::latest()->get()];
    return view('index', $context);
}
```

   Мы создаем контекст шаблона в виде ассоциативного массива и помещаем в него один элемент `bbs` — список объявлений, извлеченный из базы. При рендеринге этот элемент будет преобразован в переменную `bbs`, доступную внутри шаблона.

   Функция-хелпер `view()` готовит шаблон к рендерингу (сам рендеринг выполняется позже, непосредственно перед отправкой ответа клиенту). В первом параметре она принимает имя шаблона без расширения blade.php, а во втором — контекст шаблона. Возвращенный ею результат — подготовленный к рендерингу шаблон — следует вернуть из действия в качестве результата.

   Кстати, с функцией `view()` мы уже знакомы — видели ее в коде контроллера-функции, выводящей страницу, показанную на рис. 1.1 (см. листинг 1.2). Эта стра-

ница формируется шаблоном resources\views\welcome.blade.php, который теперь можно удалить, поскольку он больше не нужен.

2. Создадим шаблон resources\views\index.blade.php и запишем в него «заготовку» шаблона, создающего страницу с перечнем объявлений, из листинга 1.5.

Листинг 1.5. Код «заготовки» шаблона resources\views\index.blade.php

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
              content="width=device-width, initial-scale=1.0">
        <title>Объявления</title>
    </head>
    <body>
        <div class="container">
            <h1 class="my-3 text-center">Объявления</h1>
            <table class="table table-striped table-borderless">
                <thead>
                    <tr>
                        <th>Товар</th>
                        <th>Цена, руб.</th>
                        <th> </th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td><h4></h4></td>
                        <td></td>
                        <td>
                            <a href="">Подробнее...</a>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
    </body>
</html>
```

Для вывода перечня объявлений применим обычную таблицу из трех столбцов: названия товара, его цены и гиперссылки **Подробнее…**, ведущей на страницу объявления. К тегам привяжем специфические стилевые классы Bootstrap, задающие оформление для элементов страницы.

3. Перейдем на страницу **https://getbootstrap.com/docs/5.1/getting-started/introduction/**[2], найдем под заголовком **Quick start | CSS** HTML-код, привязывающий таблицу

[2] На эту страницу также можно попасть, открыв «домашний» сайт Bootstrap (**https://getbootstrap.com/**), перейдя в раздел **Docs | Getting started | Introduction** и выбрав в раскрывающемся списке в правом верхнем углу страницы пункт **Bootstrap v5.1**.

стилей Bootstrap (он содержит тег <link>), и вставим его в секцию заголовка шаблона:

```
<head>
    . . .
    <title>Объявления</title>
    <link ...>
</head>
```

4. Найдем на той же странице под заголовком **Quick start | JS | Bundle** HTML-код, привязывающий файл сценариев Bootstrap (тег <script>), и вставим его перед закрывающим тегом <body>:

```
<body>
    <div class="container">
        . . .
    </div>
    <script ... ></script>
</body>
```

В секции основного содержания таблицы (теге <tbody>) присутствует «заготовка» строки, в которой будет выводиться очередное объявление (формирующий ее тег <tr> здесь подчеркнут):

```
<tbody>
    <tr>
        <td><h4></h4></td>
        <td></td>
        <td>
            <a href="">Подробнее...</a>
        </td>
    </tr>
</tbody>
```

5. Добавим в шаблон код, который выведет строку-«заготовку» столько раз, сколько объявлений имеется в списке (как мы помним, он хранится в переменной bbs, доступной в шаблоне):

```
<tbody>
    @foreach ($bbs as $bb)
    <tr>
        . . .
    </tr>
    @endforeach
</tbody>
```

Шаблонизатор Laravel предоставляет набор своих собственных команд, аналогичных языковым конструкциям PHP и называемых *директивами*. Так, парная директива @foreach ... @endforeach аналогична циклу по массиву foreach, т. е. выводит помещенный в ней код столько раз, сколько элементов присутствует в заданном массиве или коллекции, и помещает очередной элемент в заданную переменную (у нас — bb), доступную в «теле» цикла.

6. Добавим код, выводящий название товара, цену из очередного объявления и формирующий интернет-адрес гиперссылки **Подробнее...**:

```
<tbody>
    @foreach ($bbs as $bb)
    <tr>
        <td><h4>{{ $bb->title }}</h4></td>
        <td>{{ $bb->price }}</td>
        <td>
            <a href="/{{ $bb->id }}/">Подробнее...</a>
        </td>
    </tr>
    @endforeach
</tbody>
```

Для вывода какого-либо значения следует поместить его между двойными фигурными скобками.

Осталось сделать так, чтобы, если в списке нет объявлений, таблица вообще не выводилась на странице.

7. Добавим код, выводящий таблицу лишь в том случае, если список объявлений не пуст:

```
@if (count($bbs) > 0)
<table class="table table-striped table-borderless">
    . . .
</table>
@endif
```

Парная директива `@if ... @endif` шаблонизатора аналогична по назначению условному выражению PHP.

Страница перечня объявлений готова. Приступим к странице, выводящей содержание выбранного объявления.

8. Перепишем действие `detail()` контроллера `BbsController` таким образом, чтобы оно выводило страницу, основанную на шаблоне detail.blade.php:

```
public function detail(Bb $bb) {
    return view('detail', ['bb' => $bb]);
}
```

9. Пересохраним шаблон index.blade.php под именем detail.blade.php в той же папке resources\views.

10. Запишем в шаблон detail.blade.php код, выводящий содержание выбранного объявления:

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>{{ $bb->title }} :: Объявления</title>
        . . .
    </head>
```

```
    <body>
        <div class="container">
            <h1 class="my-3 text-center">Объявления</h1>
            <h2>{{ $bb->title }}</h2>
            <p>{{ $bb->content }}</p>
            <p>{{ $bb->price }} руб.</p>
            <p><a href="/">На перечень объявлений</a></p>
        </div>
        <script . . . ></script>
    </body>
</html>
```

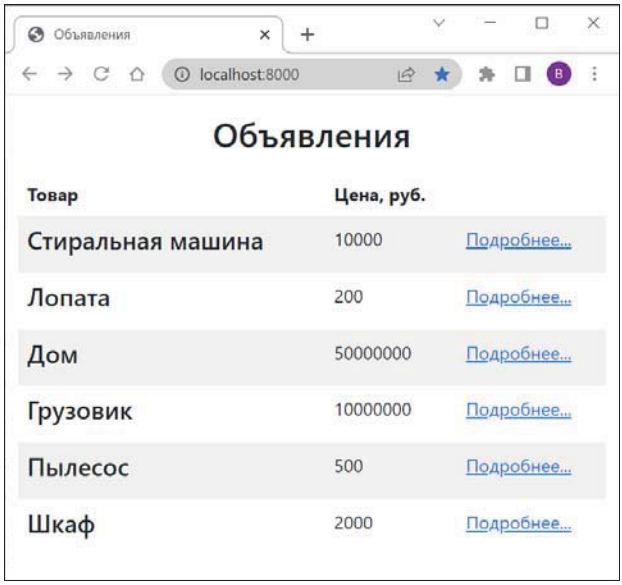


Рис. 1.5. Главная веб-страница с перечнем объявлений

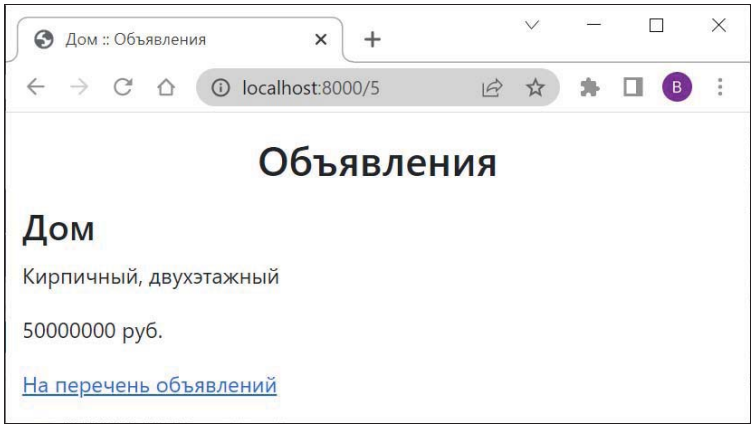


Рис. 1.6. Веб-страница с содержанием выбранного объявления

Запустим отладочный сервер и откроем сайт. Посмотрим, правильно ли выводится главная страница с перечнем объявлений (рис. 1.5). Щелкнем на любой из гиперссылок **Подробнее...** и проверим, как отображается выбранное объявление (рис. 1.6).

Наши шаблоны содержат много одинакового кода — в частности, объемистые теги `<link>` и `<script>`, привязывающие к страницам фреймворк Bootstrap. Мало того, что это увеличивает размер шаблонов и затрудняет сопровождение, но и характеризует плохой стиль программирования. Решим эту проблему.

1.13. Наследование шаблонов

Теория

Подобно классам PHP шаблоны Laravel могут наследовать друг от друга содержание (*наследование шаблонов* Laravel).

В *базовом* шаблоне (от которого выполняется наследование) записывается код, общий для всех страниц сайта: структурирующие теги, метаданные, привязки таблиц стилей и веб-сценариев, теги, формирующие разметку, шапку, поддон, панель навигации и др. *Производные* шаблоны (наследующие базовый шаблон), напротив, определяют содержание, уникальное для каждой конкретной страницы.

Фрагменты содержания в производных шаблонах оформляются в виде так называемых *секций*, имеющий уникальные имена. В базовом шаблоне помечаются места, куда будет помещено содержание той или иной секции, идентифицируемой по ее имени.

По принятому в Laravel соглашению базовые шаблоны хранятся в папке layouts, вложенной в папку resources\views.

Практика

Создадим базовый шаблон resources\views\layouts\app.blade.php[3], в который вынесем все повторяющиеся элементы из шаблонов index.blade.php и detail.blade.php. Оба этих шаблона сделаем производными от layouts\app.blade.php.

1. Создадим в папке resources\views папку layouts для хранения базового шаблона.
2. Откроем шаблон resources\views\index.blade.php (или detail.blade.php из той же папки) в текстовом редакторе и пересохраним его под именем resources\views\layouts\app.blade.php.
3. Исправим код базового шаблона resources\views\layouts\app.blade.php следующим образом:

```
<html>
    <head>
        . . .
        <title>@yield('title') :: Объявления</title>
        . . .
    </head>
```

[3] Почему именно app.blade.php, а не, например, base.blade.php, мы узнаем в *главе 2*.

```
    <body>
        <div class="container">
            <h1 class="my-3 text-center">Объявления</h1>
            @yield('content')
        </div>
        <script . . . ></script>
    </body>
</html>
```

Директива `@yield` шаблонизатора выводит на страницу содержание секции с именем, указанным в скобках. В коде шаблона мы указали места для вывода секций `title` и `content`[4], которые позже создадим в производных шаблонах и в которых запишем соответственно подзаголовок и основное содержание страницы.

4. Откроем шаблон resources\views\index.blade.php и превратим его в производный от только что написанного базового шаблона, внеся следующие правки:

```
@extends('layouts.app')

@section('title', 'Главная')

@section('content')
@if (count($bbs) > 0)
<table class="table table-striped">
    . . .
</table>
@endif
@endsection('content')
```

Директива `@extends` указывает базовый шаблон, от которого наследует текущий. Для разделения имен папок и файлов в путях к шаблонам используются не слеши, а точки.

Директива `@section` создает секцию, чье имя указано в первом параметре. Она может быть записана в двух форматах:

- как одинарная — тогда содержание секции задается вторым параметром;
- как парная (`@section ... @endsection`) — тогда содержание помещается в тело директивы.

5. Внесем аналогичные правки в шаблон resources\views\detail.blade.php:

```
@extends('layouts.app')

@section('title', $bb->title)
@section('content')
<h2>{{ $bb->title }}</h2>
. . .
<p><a href="/">На перечень объявлений</a></p>
@endsection('content')
```

Проверим сайт в работе и убедимся, что все страницы выводятся правильно.

---

[4] Почему именно `content`, а не, скажем, `main`, также будет объяснено в *главе 2*.

Наш сайт имеет недостаток: интернет-адреса в гиперссылках генерируются непосредственно в программном коде. Например, интернет-адреса страниц объявлений формата **/*<ключ объявления>*/** генерируются так (выводящий их код подчеркнут):

```
<a href="/{{ $bb->id }}/">Подробнее...</a>
```

Если мы решим поменять формат этих адресов, скажем, на **/detail/*<ключ объявления>*/**, то будем вынуждены вносить правки во все шаблоны, в которых присутствуют гиперссылки с такими адресами. Решим и эту проблему.

1.14. Именованные маршруты

Именованный маршрут — маршрут, которому дано уникальное имя. Laravel может генерировать интернет-адреса формата, задаваемого именованным маршрутом, для чего достаточно задать имя маршрута и значения URL-параметров (если таковые имеются в шаблонном пути).

Дадим имена маршрутам:

- ведущему на главную страницу — имя `index`;
- ведущему на страницу объявления — имя `detail`.

И сделаем так, чтобы интернет-адреса гиперссылок генерировались на основании имен маршрутов.

1. Откроем модуль routes\web.php, хранящий список веб-маршрутов, и укажем имена у маршрутов, добавив следующий код:

   ```
   Route::get('/', [BbsController::class, 'index'])->name('index');
   Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
   ```

   Метод `get()` маршрутизатора в качестве результата возвращает объект, представляющий маршрут. Метод `name()`, вызванный у этого объекта, дает маршруту заданное в параметре имя.

2. Откроем модуль resources\views\index.blade.php, где хранится шаблон перечня объявлений, и исправим код, выводящий гиперссылки на страницы объявлений:

   ```
   <td>
       <a href="{{ route('detail', ['bb' => $bb->id]) }}"> ... </a>
   </td>
   ```

   Функция-хелпер `route()` генерирует интернет-адрес на основе именованного маршрута, имя которого задано в первом параметре. Во втором параметре может быть указан ассоциативный массив со значениями URL-параметров.

3. Откроем модуль resources\views\detail.blade.php с шаблоном выбранного объявления и исправим код гиперссылки на перечень объявлений:

   ```
   <p><a href="{{ route('index') }}">На перечень объявлений</a></p>
   ```

Проверим сайт в работе. И напоследок немного разукрасим его.

1.15. Статические файлы

Статическими называются файлы, пересылаемые клиенту как есть, без какой бы то ни было обработки: внешние таблицы стилей, веб-сценарии, изображения, аудио- и видеоролики, документы, архивы и пр.

Статические файлы Laravel-сайта располагаются в папке public, фактически являющейся корневой папкой этого сайта, и во вложенных в нее папках.

Выведем в заголовке сайта, слева и справа, две копии небольшого графического изображения. Само изображение сохраним в файле public\images\logo.jpg, а таблицу стилей, задающую оформление, — в файле public\styles\main.css.

1. Создадим в папке public папки images и styles.
2. Найдем в Интернете подходящее изображение и сохраним его под именем logo.jpg в папке public\images.

   Если найденное изображение записано в графическом формате, отличном от JPEG, следует дать файлу соответствующее расширение.
3. Создадим в папке public\styles файл main.css и запишем в него код таблицы стилей из листинга 1.6.

Листинг 1.6. Код таблицы стилей public\styles\main.css

```
h1 {
    background: url("/images/logo.jpg") left / auto 100% no-repeat,
                url("/images/logo.jpg") right / auto 100% no-repeat;
}
```

   Если файл с изображением имеет расширение, отличное от jpg, CSS-код следует соответственно исправить.

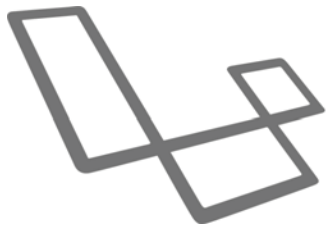
   Абсолютный интернет-адрес файла с изображением указывается относительно папки public.
4. Откроем модуль resources\views\layouts\app.blade.php, содержащий код базового шаблона, и вставим в секцию заголовка тег <link>, привязывающий нашу таблицу стилей:

```
<head>
    . . .
    <link ... >
    <link href="/styles/main.css" rel="stylesheet" type="text/css">
</head>
```

Теперь наша доска объявлений выглядит симпатичнее — рис. 1.7.

В следующей главе мы продолжим совершенствовать сайт, добавив средства разграничения доступа, создания, правки и удаления объявлений.

Рис. 1.7. Заголовок веб-сайта после применения стилей



ГЛАВА 2

Доска объявлений 2.0: разграничение доступа, работа с объявлениями и локализация

В этой главе мы свяжем каждое объявление с определенным зарегистрированным пользователем, добавим инструменты для создания, правки и удаления объявлений, регистрации пользователей, защитим сайт от несанкционированного доступа и полностью переведем его на русский язык.

2.1. Межтабличные связи. Работа со связанными записями

Создание связи между таблицами выполняется в три этапа:

- формирование поля, хранящего ключ связанной записи первичной таблицы (*поля внешнего ключа*), — во вторичной таблице;
- объявление «прямой» связи (между первичной и вторичной моделями) — в модели первичной таблицы;
- объявление «обратной» связи (между вторичной и первичной моделями) — в модели вторичной таблицы.

Свяжем каждое объявление с определенным пользователем-автором, для чего установим связь «один-со-многими» между таблицами: `users` (она станет первичной) и `bbs` (а она — вторичной). Далее добавим в список нового пользователя, создадим разными способами три связанных с ним объявления и реализуем вывод на странице объявления имени его автора.

Чтобы избежать проблем с заполнением поля вторичного ключа в таблице `bbs`, не будем вносить правки в уже имеющуюся таблицу, а удалим ее и создадим заново. Для удаления таблицы `bbs` достаточно откатить последнюю миграцию.

1. В командной строке выполним откат последней миграции:

   ```
   php artisan migrate:rollback --step=1
   ```

   Количество откатываемых миграций указывается в параметре `--step` команды `migrate:rollback`.

2. Откроем в текстовом редакторе модуль database\migrations\<*текущая временна́я отметка*>_create_bbs_table.php с нашей миграцией и добавим код, создающий поле внешнего ключа:

```
public function up() {
    Schema::create('bbs', function (Blueprint $table) {
        . . .
        $table->float('price');
        $table->foreignId('user_id')->constrained()
              ->onDelete('cascade');
        $table->timestamps();
        . . .
    });
}
```

Метод `foreignId()` объекта структуры таблицы создает поле внешнего ключа и возвращает представляющий его объект. Мы дали этому полю «говорящее» имя `user_id` и поэтому для создания собственно связи можем вызвать у возвращенного объекта связи метод `constrained()`, который извлечет из имени все необходимые ему сведения: имя связываемой первичной таблицы (`users`) и ее ключевое поле (`id`). Метод `onDelete()` укажет операцию, выполняемую над связанными записями вторичной таблицы при удалении записи первичной таблицы (у нас — `cascade`, т. е. каскадное удаление).

3. В командной строке применим исправленную миграцию:

```
php artisan migrate
```

Исправленная таблица объявлений готова. Займемся моделями пользователей и объявлений.

4. Откроем в текстовом редакторе модуль app\Models\User.php с классом модели пользователя `User` и добавим код, объявляющий «прямую» связь между текущей, первичной, и заданной вторичной моделями:

```
use App\Models\Bb;
class User extends Authenticatable {
    . . .
    public function bbs() {
        return $this->hasMany(Bb::class);
    }
}
```

«Прямая» связь объявляется в виде обычного общедоступного метода (у нас — `bbs()`). В нем вызывается метод `hasMany()` модели, принимающий имя класса связываемой вторичной модели и возвращающий объект созданной связи. Последний следует вернуть из метода, объявляющего связь.

5. Откроем в текстовом редакторе модуль app\Models\Bb.php с классом модели объявления `Bb` и добавим код, объявляющий «обратную» связь между текущей, вторичной, и заданной первичной моделями:

```
use App\Models\User;
class Bb extends Model {
    . . .
```

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

«Обратная» связь также объявляется в виде общедоступного метода (у нас — `user()`). Метод `belongsTo()` класса модели принимает имя класса связываемой первичной модели и возвращает объект созданной связи, который следует вернуть из метода, объявляющего связь.

6. Запустим консоль Laravel (см. *разд. 1.9*) и добавим нового пользователя с именем `admin`, адресом электронной почты **admin@bboard.ru** (адрес можно указать произвольный, поскольку Laravel по умолчанию не проверяет его существование) и паролем `admin` (задается в виде хеша, вычисленного вызовом метода `make()` у подсистемы хеширования, представляемой фасадом `Hash`):

```
>>> use Illuminate\Support\Facades\Hash;
>>> use App\Models\User;
>>> $user = User::create(['name' => 'admin',
...                       'email' => 'admin@bboard.ru',
...                       'password' => Hash::make('admin')]);
```

7. Добавим объявление от имени этого пользователя:

```
>>> use App\Models\Bb;
>>> $bb = new Bb();
>>> $bb->title = 'Пылесос';
>>> $bb->content = 'Старый, ржавый, без шланга';
>>> $bb->price = 500;
>>> $user->bbs()->save($bb);
```

Метод первичной модели, объявляющий «прямую» связь (у нас — `bbs()`), возвращает объект «прямой» связи. Последний поддерживает метод `save()`, связывающий переданную в параметре запись вторичной модели с текущей записью первичной модели и одновременно сохраняющий переданную запись.

8. Создадим еще одно объявление другим способом:

```
>>> $user->bbs()->create(['title' => 'Грузовик',
...                       'content' => 'Грузоподъемность - 5 т',
...                       'price' => 10000000]);
```

А еще объект «прямой» связи поддерживает метод `create()`, знакомый нам по *разд. 1.10*.

9. Добавим третье объявление — третьим способом:

```
>>> $bb = new Bb(['title' => 'Шкаф',
...       'content' => 'Совсем новый, полированный, двухстворчатый',
...       'price' => 1000]);
>>> $bb->user()->associate($user);
>>> $bb->save();
```

Конструктору модели можно передать ассоциативный массив со значениями полей (как и методу `create()`).

Метод вторичной модели, объявляющий «обратную» связь (у нас — `user()`), возвращает объект «обратной» связи. Последний поддерживает метод `associate()`, связывающий переданную в параметре запись первичной модели с текущей записью вторичной модели. Однако этот метод не сохраняет текущую запись, и нам придется сделать это самостоятельно, вызвав метод `save()`.

10. Переберем все объявления и для каждого выведем название товара, имя пользователя-автора и цену:

```
>>> foreach (Bb::all() as $bb) {
...   $user = $bb->user;
...   echo $bb->title, ' | ', $user->name, ' | ', $bb->price, "\r\n";
... }
Пылесос | admin | 500.0
Грузовик | admin | 10000000.0
Шкаф | admin | 1000.0
```

Метод `all()`, вызываемый у модели (но выполняемый построителем запросов), возвращает все записи таблицы. Свойство `user` объявления (одноименное с ранее объявленным в модели `Bb` методом) хранит связанного с объявлением пользователя.

11. Переберем все объявления, оставленные пользователем `admin`, и выведем названия хранящихся в них товаров:

```
>>> $user = User::where('name' ,'admin')->first();
>>> foreach ($user->bbs as $bb) { echo $bb->title, ' '; }
Пылесос Грузовик Шкаф
```

Свойство `bbs` пользователя (одноименное с ранее объявленным в модели `User` методом) хранит список связанных объявлений.

12. Откроем модуль resources\views\detail.blade.php, хранящий код шаблона объявления, и добавим вывод имени пользователя, оставившего объявление:

```
<p>{{ $bb->price }} руб.</p>
<p>Автор: {{ $bb->user->name }}</p>
<p><a href="{{ route('index') }}">На перечень объявлений</a></p>
```

Проверим сайт в работе. И приступим к реализации входа на сайт, раздела пользователя и выхода с сайта.

2.2. Вход и выход. Раздел пользователя

Теория

К работе с внутренними данными сайта (в нашем случае — со списком объявлений) следует допускать только посетителей, указанных в особом списке, — *зарегистрированных пользователей*, или просто *пользователей*.

Список пользователей практически всегда хранится в одной из таблиц базы данных. Каждая позиция списка содержит внутреннее имя пользователя («логин»), адрес его электронной почты, пароль (в закодированном виде) и, возможно, дополнительные

сведения: настоящие имя и фамилию пользователя, наименования операций, которые пользователь может выполнять с данными (*привилегии*, или *права*), и др.

Посетитель, желающий получить доступ к данным сайта, должен выполнить процедуру *входа* на сайт (или *аутентификации*). Для этого он переходит на *веб-страницу входа* и вводит в веб-форму свои адрес электронной почты и пароль. Laravel находит в списке пользователя с заданными адресом и паролем и помечает его как выполнившего вход — *текущего* пользователя (записывая в серверную сессию особый признак).

При попытке перейти на какую-либо страницу Laravel проверяет, кому должна быть доступна эта страница: всем, только зарегистрированным пользователям, выполнившим вход, только зарегистрированным пользователям с особыми привилегиями или только посетителям, не выполнившим вход (*гостям*). Если целевая страница не должна быть доступна текущему пользователю, выполняется перенаправление на страницу входа (если вход на сайт не был выполнен) или выдается сообщение об ошибке 403 (пользователь не имеет необходимых прав) — если вход на сайт был выполнен. Такого рода проверки называются *разграничением доступа* (или *авторизацией*).

Раздел пользователя — это страница, выводящая какие-либо данные, которые принадлежат текущему пользователю (например, перечень принадлежащих ему объявлений). Обычно именно в этот раздел производится перенаправление после успешного входа на сайт.

Закончив работу, пользователь выполняет *выход* с сайта. При этом фреймворк «забывает», что пользователь ранее выполнил вход, и начинает считать его гостем.

Библиотека laravel/ui, установленная в *разд. 1.2*, позволяет сформировать в разрабатываемом сайте базовые средства для разграничения доступа: контроллеры, реализующие вход, раздел пользователя и выход, а также необходимые маршруты и шаблоны страниц.

Любой Laravel-проект изначально содержит миграцию, создающую таблицу `users`, которая хранит список зарегистрированных пользователей. Соответственно, эта таблица создается при первом выполнении миграций (см. *разд. 1.7*) — нам самим формировать ее не придется.

Практика

Создадим «костяк» подсистемы разграничения доступа и раздел пользователя, выводящий объявления, чьим автором является текущий пользователь.

1. В командной строке создадим базовые средства разграничения доступа:

   ```
   php artisan ui:auth
   ```

   В консоли появится сообщение:

   ```
   The [layouts/app.blade.php] view already exists. Do you want to
   replace it? (yes/no) [no]:
   ```

   Перевод:

   ```
   Шаблон layouts\abb.blade.php уже существует. Вы хотите перезаписать
   его? (да/нет) [Нет]:
   ```

   Нам не нужно, чтобы наш базовый шаблон был перезаписан.

2. Чтобы отменить перезапись базового шаблона — введем строку `no`, букву `n` или просто нажмем клавишу <Enter>.

   После выполнения команды `ui:auth` будут созданы, в частности, следующие модули:

   - контроллеры (имена классов указаны относительно пространства имен `App\Http\Controllers`):
     - `HomeController` — выводит раздел пользователя;
     - `Auth\RegisterController` — регистрирует нового пользователя;
     - `Auth\LoginController` — выполняет вход на сайт и выход с него;
   - шаблоны (пути указаны относительно папки resources\views):
     - home.blade.php — шаблон страницы с разделом пользователя;
     - auth\register.blade.php — шаблон страницы регистрации нового пользователя;
     - auth\login.blade.php — шаблон страницы входа.

   Также будут созданы еще несколько контроллеров (и соответствующих им шаблонов), выполняющих проверку, сброс пароля и активацию нового пользователя по электронной почте. Мы рассмотрим их в *главе 13*.

   В список веб-маршрутов будут добавлены маршруты, ведущие на действия вновь созданных контроллеров.

3. Откроем модуль routes\web.php со списком веб-маршрутов и посмотрим, какие выражения были добавлены в него (здесь они подчеркнуты):

```
Route::get('/', [BbsController::class, 'index'])->name('index');
Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
Auth::routes();
Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])
    ->name('home');
```

   Метод `routes()` фасада `Auth` (за ним «прячется» подсистема безопасности) создает маршруты на действия контроллеров, созданных командой `ui:auth` утилиты artisan. Из второго добавленного выражения видно, что раздел пользователя выводится действием `index()` контроллера `HomeController` при запросе по пути **/home/** методом GET, а соответствующий маршрут имеет имя `home` — запомним его.

4. В командной строке выведем список маршрутов, созданных методом `routes()` фасада `Auth`:

```
php artisan route:list
```

   Из всех представленных в выведенном списке маршрутов нас интересуют лишь следующие (приведены в формате: «допустимый HTTP-метод — действие — описание»):

   - GET — `register` — вывод страницы регистрации;
   - GET — `login` — вывод страницы входа;
   - POST — `logout` — выполнение выхода.

Полученных сведений достаточно для создания гиперссылок. Затруднения возникнут лишь с маршрутом `logout`, поскольку в нем указан допустимый метод POST. Но мы избежим их, создав веб-форму с кнопкой, отправляющую данные по этому маршруту методом POST.

5. Откроем модуль resources\views\layouts\app.blade.php с базовым шаблоном и добавим в него горизонтальную панель навигации:

```
<div class="container">
    <nav class="navbar navbar-light bg-light">
        <div class="container-fluid">
            <a href="{{ route('index') }}"
               class="navbar-brand me-auto">Главная</a>
            <a href="{{ route('register') }}"
               class="nav-item nav-link">Регистрация</a>
            <a href="{{ route('login') }}"
               class="nav-item nav-link">Вход</a>
            <a href="{{ route('home') }}"
               class="nav-item nav-link">Мои объявления</a>
            <form action="{{ route('logout') }}" method="POST"
                  class="form-inline">
                @csrf
                <input type="submit" class="btn btn-danger"
                                     value="Выход">
        </form>
        </div>
    </nav>
    <h1 class="my-3 text-center">Объявления</h1>
    @yield('content')
</div>
```

Директива `@csrf` шаблонизатора вставляет в форму скрытое поле с электронным маркером безопасности (подробности — в *главе 11*). Если мы не вставим этот маркер, то получим сообщение об ошибке 419 (страница устарела).

В *разд. 1.13*, создавая базовый шаблон, мы дали ему имя layouts\app.blade.php, а для размещения основного содержания предусмотрели в шаблоне секцию с именем `content`. Сделано это было не просто так. Базовый шаблон, создаваемый командой `ui:auth`, также имеет имя layouts\app.blade.php, а основное содержание в нем также размещается в секции `content`. Так что нам придется внести в производные шаблоны совсем небольшие правки.

6. Откроем модуль resources\views\auth\login.blade.php, хранящий шаблон страницы входа, и внесем следующую правку:

```
@extends('layouts.app')

@section('title', 'Вход')
. . .
```

Мы лишь добавили секцию `title` с названием подраздела сайта.

7. Откроем модуль resources\views\auth\register.blade.php, хранящий шаблон страницы регистрации, и внесем аналогичную правку.

Займемся разделом пользователя, который будет выводить перечень объявлений, оставленных текущим пользователем, в обратном хронологическом порядке.

8. Откроем модуль app\Http\Controllers\HomeController.php с кодом контроллера `HomeController` и внесем следующие правки:

```
use Illuminate\Support\Facades\Auth;
class HomeController extends Controller {
    . . .
    public function index() {
        return view('home',
                    ['bbs' => Auth::user()->bbs()->latest()->get()]);
    }
}
```

Метод `user()` фасада `Auth` возвращает объект модели `User`, представляющий текущего пользователя. Вызвав у пользователя метод `bbs()`, получим объект «прямой» связи. Этот объект имеет функциональность построителя запросов, поддерживает все его методы (`latest()`, `where()`, `get()` и др., подробности — в *разд. 1.10*) и настроен на обработку только связанных записей.

9. Откроем модуль resources\views\home.blade.php, где записан шаблон страницы с разделом пользователя, удалим весь имеющийся там код и запишем в него код, показанный в листинге 2.1.

Листинг 2.1. Код шаблона resources\views\home.blade.php

```
@extends('layouts.app')

@section('title', 'Мои объявления')

@section('content')
<p class="text-end"><a href="">Добавить объявление</a></p>
@if (count($bbs) > 0)
<table class="table table-striped table-borderless">
    <thead>
        <tr>
            <th>Товар</th>
            <th>Цена, руб.</th>
            <th colspan="2"> </th>
        </tr>
    </thead>
    <tbody>
        @foreach ($bbs as $bb)
        <tr>
            <td><h3>{{ $bb->title }}</h3></td>
            <td>{{ $bb->price }}</td>
            <td>
                <a href="">Изменить</a>
            </td>
```

```
            <td>
                <a href="">Удалить</a>
            </td>
        </tr>
        @endforeach
    </tbody>
</table>
@endif
@endsection('content')
```

Мы сразу создали гиперссылки для добавления, правки и удаления объявлений. Соответствующие интернет-адреса занесем в них позже.

10. Запустим отладочный сервер, откроем сайт и попытаемся перейти на страницу входа, щелкнув на гиперссылке **Вход**.

    Результатом станет стандартная страница с сообщением об ошибке 404, выводимая Laravel. Мы где-то допустили ошибку...

    Вернемся к списку маршрутов:

```
Route::get('/', [BbsController::class, 'index'])->name('index');
Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
Auth::routes();
Route::get('/home', [App\Http\Controllers\HomeController::class,
           'index'])
    ->name('home');
```

    Маршрутизатор Laravel при просмотре списка маршрутов выбирает *самый первый* маршрут, имеющий совпадающие шаблонный путь и допустимый HTTP-метод. Остальные маршруты при этом не просматриваются.

    У нас страница входа располагается по пути **/login/**. Просматривая список маршрутов, маршрутизатор обнаружит, что этот путь совпадает с шаблонным путем из второго по счету маршрута, ведущего на действие `detail()` контроллера `BbsController` (оно выводит страницу объявления). Попытка извлечь объявление с ключом `login` увенчается неудачей и генерированием исключения, выводящего сообщение об ошибке 404.

    Чтобы устранить проблему, достаточно передвинуть маршрут, ведущий на действие `detail()` контроллера `BbsController`, в самый конец списка.

11. Откроем модуль routes\web.php со списком веб-маршрутов и внесем в него нужные правки:

```
Route::get('/', [BbsController::class, 'index'])->name('index');
~~Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');~~
Auth::routes();
Route::get('/home', [App\Http\Controllers\HomeController::class,
           'index'])
    ->name('home');
Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
```

Теперь все будет работать. Откроем сайт, перейдем на страницу входа (рис. 2.1), укажем адрес электронной почты **admin@bboard.ru**, пароль `admin` и нажмем кнопку

Login. Если мы не допустили ошибок при вводе, окажемся на странице раздела пользователя (рис. 2.2). Напоследок выйдем с сайта, нажав кнопку **Выход**.

Полезно знать

По умолчанию Laravel идентифицирует пользователя по адресу его электронной почты. Подавляющее большинство других фреймворков используют для этого регистрационное имя пользователя («логин»).

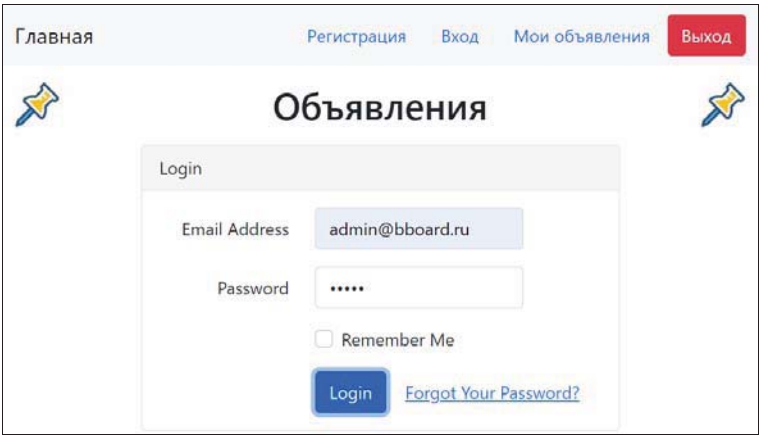


Рис. 2.1. Веб-страница входа

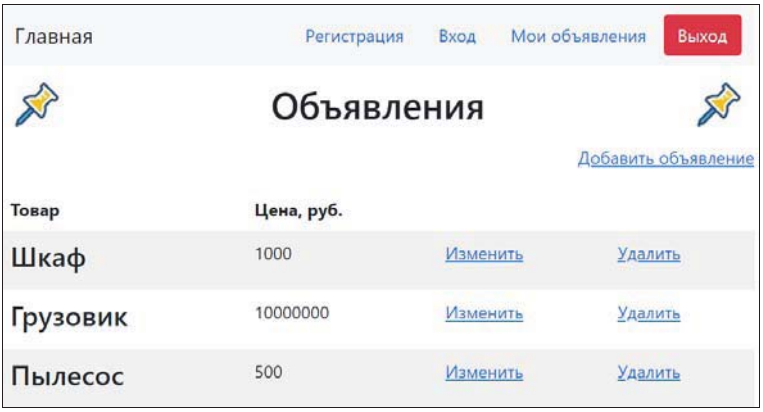


Рис. 2.2. Веб-страница раздела пользователя

Пока еще все надписи на страницах регистрации и входа выводятся по-английски. В конце этой главы мы выполним их локализацию на русский язык.

2.3. Добавление, правка и удаление записей

Наделим наш сайт средствами для добавления новых объявлений. Автором добавляемых объявлений станет текущий пользователь. Позже реализуем правку и удаление объявлений. Всю необходимую логику запишем в контроллер `HomeController`.

1. Откроем список веб-маршрутов routes\web.php и добавим два маршрута, необходимых для реализации добавления объявлений:

```
use App\Http\Controllers\HomeController;
. . .
Route::get('/home', [HomeController::class, 'index'])->name('home');
Route::get('/home/create', [HomeController::class, 'create'])
     ->name('bb.create');
Route::post('/home', [HomeController::class, 'store'])
     ->name('bb.store');
Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
```

   Первый из добавленных маршрутов (с шаблонным путем **/home/create/** и допустимым методом GET) связан с действием `create()` контроллера `HomeController` и выведет страницу с веб-формой для занесения нового объявления. Второй маршрут (шаблонный путь — **/home/**, допустимый HTTP-метод — POST, создан вызовом метода `post()`) связан с действием `store()` того же контроллера, добавляющим введенное объявление в базу.

2. Откроем контроллер app\Http\Controllers\HomeController.php и добавим в него действие `create()`, выводящее страницу добавления объявления:

```
class HomeController extends Controller {
    . . .
    public function create() {
        return view('bb-create');
    }
}
```

   Шаблон bb-create.blade.php, формирующий страницу для ввода объявления, мы создадим позже. На этой странице предусмотрим поле ввода с наименованием `title`, область редактирования `content` и поле ввода `price`.

3. Добавим в контроллер `HomeController` действие `store()`, сохраняющее новое объявление и выполняющее перенаправление на раздел пользователя:

```
class HomeController extends Controller {
    . . .
    public function store(Request $request) {
        Auth::user()->bbs()->create(['title' => $request->title,
                                     'content' => $request->content,
                                     'price' => $request->price]);
        return redirect()->route('home');
    }
}
```

   Мы задали у действия `store()` параметр типа `Request`, тем самым «намекнув» подсистеме внедрения зависимостей Laravel, что хотим получить в этом параметре объект класса `Request`, представляющий запрос. Из его свойств, чьи имена совпадают с наименованиями элементов управления, извлекаем занесенные в веб-форму значения для сохранения в записи.

   Далее, вызвав функцию-хелпер `redirect()`, получаем «пустой» объект перенаправления и, вызвав у этого объекта метод `route()`, заносим в него целевой интернет-

адрес, сгенерированный на основе именованного маршрута `home`. И не забываем вернуть из действия готовый объект перенаправления.

4. Создадим модуль resources\views\bb-create.blade.php и запишем в него код шаблона страницы для ввода нового объявления из листинга 2.2.

Листинг 2.2. Код шаблона resources\views\bb-create.blade.php

```
@extends('layouts.app')

@section('title', 'Добавление объявления :: Мои объявления')

@section('content')
<form action="{{ route('bb.store') }}" method="POST">
    @csrf
    <div class="mb-3">
        <label for="txtTitle" class="form-label">Товар</label>
        <input name="title" id="txtTitle" class="form-control">
    </div>
    <div class="mb-3">
        <label for="txtContent" class="form-label">Описание</label>
        <textarea name="content" id="txtContent" class="form-control"
                  row="3"></textarea>
    </div>
    <div class="mb-3">
        <label for="txtPrice" class="form-label">Цена</label>
        <input name="price" id="txtPrice" class="form-control">
    </div>
    <input type="submit" class="btn btn-primary" value="Добавить">
</form>
@endsection('content')
```

Не забываем поместить в форму электронный маркер безопасности, воспользовавшись директивой `@csrf`.

5. Откроем шаблон раздела пользователя resources\views\home.blade.php и запишем интернет-адрес в гиперссылку **Добавить объявление**:

```
<p class="text-right"><a href="{{ route('bb.create') }}">Добавить
   объявление</a></p>
```

6. Запустим отладочный сервер, выполним вход на сайт от имени пользователя **admin@bboard.ru**, перейдем на страницу добавления объявления (рис. 2.3) и создадим объявление с произвольным содержанием. Затем выйдем с сайта, перейдем на страницу регистрации и создадим еще одного пользователя с именем `editor`, адресом **editor@bboard.ru** и паролем `supereditor`. После создания пользователя Laravel автоматически выполнит вход на сайт от его имени. Добавим еще пару произвольных объявлений от имени нового пользователя и выйдем с сайта.

   Наконец, дадим пользователям возможность править и удалять объявления.

7. Откроем список веб-маршрутов routes\web.php и добавим еще четыре маршрута:

```
Route::post('/home', [HomeController::class, 'store'])
    ->name('bb.store');
Route::get('/home/{bb}/edit', [HomeController::class, 'edit'])
    ->name('bb.edit');
Route::patch('/home/{bb}', [HomeController::class, 'update'])
    ->name('bb.update');
Route::get('/home/{bb}/delete', [HomeController::class, 'delete'])
    ->name('bb.delete');
Route::delete('/home/{bb}', [HomeController::class, 'destroy'])
    ->name('bb.destroy');
Route::get('/{bb}', [BbsController::class, 'detail'])->name('detail');
```

Первый добавленный маршрут выведет страницу с веб-формой для правки объявления, второй — сохранит исправленное объявление, третий — выведет страницу удаления объявления, четвертый — удалит объявление. У второго маршрута в качестве допустимого HTTP-метода указан PATCH (вызовом метода `patch()` у фасада `Route`), а у четвертого — DELETE (вызовом метода `delete()`).

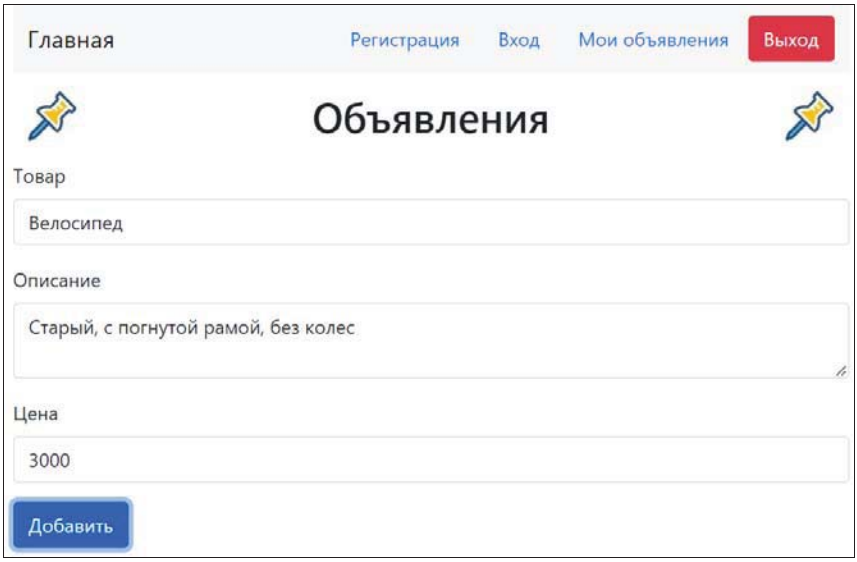


Рис. 2.3. Веб-страница добавления объявления

8. Откроем контроллер app\Http\Controllers\HomeController.php и добавим в него действия: `edit()` (вывод страницы правки объявления) и `update()` (сохранение исправленного объявления):

```
use App\Models\Bb;
class HomeController extends Controller {
    . . .
    public function edit(Bb $bb) {
        return view('bb-edit', ['bb' => $bb]);
    }
```

```
    public function update(Request $request, Bb $bb) {
        $bb->fill(['title' => $request->title,
                   'content' => $request->content,
                   'price' => $request->price]);
        $bb->save();
        return redirect()->route('home');
    }
}
```

В действии `edit()` не забываем занести в контекст шаблона выбранную запись — чтобы подставить значения ее полей в поля ввода веб-формы.

В действии `update()` мы указали два параметра: типа `Request` — для объекта запроса, и типа `Bb` — для объекта модели, представляющего выбранное объявление. Подсистема внедрения зависимостей Laravel сама занесет в них требуемые значения.

Для массового присваивания исправленных значений полям объявления применяем метод `fill()` объекта модели. Сохраняем запись вызовом метода `save()`.

9. Добавим в контроллер `HomeController` действия: `delete()` (вывод страницы удаления объявления) и `destroy()` (собственно удаление):

```
class HomeController extends Controller {
    . . .
    public function delete(Bb $bb) {
        return view('bb-delete', ['bb' => $bb]);
    }

    public function destroy(Bb $bb) {
        $bb->delete();
        return redirect()->route('home');
    }
}
```

В действии `delete()` заносим в контекст шаблона выбранную запись — чтобы вывести ее на странице.

10. Пересохраним модуль resources\views\bb-create.blade.php под именем bb-edit.blade.php, тем самым создав шаблон страницы для правки объявления, и исправим его код:

```
@section('title', 'Правка объявления :: Мои объявления')

@section('content')
<form action="{{ route('bb.update', ['bb' => $bb->id]) }}"
      method="POST">
    @csrf
    @method('PATCH')
    <div class="mb-3">
        . . .
        <input ... value="{{ $bb->title }}">
    </div>
    <div class="mb-3">
        . . .
```

```
            <textarea ... >{{ $bb->content }}</textarea>
        </div>
        <div class="mb-3">
            . . .
            <input ... value="{{ $bb->price }}">
        </div>
        <input ... value="Сохранить">
    </form>
    @endsection('content')
```

В маршруте, сохраняющем исправленное объявление, мы указали допустимый HTTP-метод PATCH. Проблема в том, что он не поддерживается веб-обозревателями, — мы не можем записать его в атрибуте `method` тега `<form>`. Однако можно поместить в форму скрытое поле с наименованием нужного HTTP-метода. Это выполняется директивой `@method` шаблонизатора.

В поля ввода и область редактирования следует подставить значения полей исправляемой записи.

11. Пересохраним модуль resources\views\detail.blade.php под именем bb-delete.blade.php, создав шаблон страницы для удаления объявления, и исправим его код:

```
@section('title', 'Удаление объявления :: Мои объявления')

@section('content')
. . .
<p>Автор: {{ $bb->user->name }}</p>
<form action="{{ route('bb.destroy', ['bb' => $bb->id]) }}"
      method="POST">
    @csrf
    @method('DELETE')
    <input type="submit" class="btn btn-danger" value="Удалить">
</form>
@endsection('content')
```

Чтобы отправить запрос на удаление объявления HTTP-методом DELETE (который также не поддерживается веб-обозревателями), мы создали на странице веб-форму с кнопкой отправки данных и с помощью директивы `@method` поместили в форму скрытое поле с наименованием нужного HTTP-метода.

12. Откроем шаблон раздела пользователя resources\views\home.blade.php и вставим интернет-адреса в гиперссылки **Изменить** и **Удалить**:

```
<a href="{{ route('bb.edit', ['bb' => $bb->id]) }}">Изменить</a>
. . .
<a href="{{ route('bb.delete', ['bb' => $bb->id]) }}">Удалить</a>
```

Войдем на сайт от имени любого из зарегистрированных пользователей и попробуем исправить какое-либо объявление. Добавим произвольное объявление и попытаемся удалить его.

На этом этапе сайт еще не проверяет корректность заносимых данных. Например, если в поле ввода **Цена** вместо числа ввести строку, она будет успешно сохранена в базе (формат базы данных SQLite позволяет хранить в поле значения любого типа, даже не

совпадающего с типом, указанным при объявлении поля). А если вообще не заносить значение в какое-либо поле, мы получим страницу с системным сообщением об ошибке, которое ничего не скажет конечному пользователю сайта (зато многое скажет потенциальному взломщику).

Поэтому самое время заняться валидацией.

2.4. Валидация данных

Валидация — это проверка данных на корректность на основе заданных *правил*.

Реализуем валидацию добавляемых и исправляемых объявлений согласно следующим правилам:

- поле `title` — обязательно к заполнению, максимальная длина значения 50 символов;
- поле `content` — обязательно к заполнению;
- поле `price` — обязательно к заполнению, значение должно быть числом.

1. Откроем модуль app\Http\Controllers\HomeController.php и объявим в классе контроллера `HomeController` константу `BB_VALIDATOR` с перечнем правил валидации:

```
class HomeController extends Controller {
    private const BB_VALIDATOR = [
        'title' => 'required|max:50',
        'content' => 'required',
        'price' => 'required|numeric'
    ];
    . . .
}
```

   Набор правил валидации записывается в виде ассоциативного массива, каждый элемент которого задает набор правил для отдельного элемента управления. Отдельные правила в наборе разделяются символом вертикальной черты. У поля ввода `title` мы указали правила `required` (обязательно к заполнению) и `max:50` (длина значения — не более 50 символов), у области редактирования `content` — `required`, у поля ввода `price` — `required` и `numeric` (значение должно представлять собой число).

2. Добавим в действия `store()` и `update()` того же контроллера код, выполняющий валидацию согласно записанным ранее правилам:

```
public function store(Request $request) {
    $validated = $request->validate(self::BB_VALIDATOR);
    Auth::user()->bbs()->create(['title' => $validated['title'],
                                 'content' => $validated['content'],
                                 'price' => $validated['price']]);
    return redirect()->route('home');
}
. . .
public function update(Request $request, Bb $bb) {
    $validated = $request->validate(self::BB_VALIDATOR);
```

```
        $bb->fill(['title' => $validated['title'],
                   'content' => $validated['content'],
                   'price' => $validated['price']]);
        $bb->save();
        . . .
    }
```

Валидация запускается вызовом метода `validate()` объекта запроса, в котором и содержатся подлежащие валидации данные из веб-формы. В качестве параметра этому методу передаются перечень правил валидации.

Если валидация прошла успешно, метод `validate()` вернет ассоциативный массив с данными из веб-формы, очищенными от лишних символов (начальных и конечных пробелов, букв, поставленных после цифр в числах, и др.). Данные из этого массива мы и сохраняем в базе.

Если валидация не увенчалась успехом, Laravel сохранит в серверной сессии все данные из веб-формы и все сообщения об ошибках ввода, после чего выполнит перенаправление на предыдущую страницу — ту самую, на которой находится веб-форма. В результате посетитель сможет исправить некорректные данные.

Теперь нужно сделать так, чтобы на страницах с веб-формами добавления и правки объявления выводились сообщения об ошибках ввода и ранее введенные данные.

3. Откроем шаблон страницы правки объявления resources\views\bb-edit.blade.php и добавим код, выводящий сообщения об ошибках ввода в поле `title`:

```
<div class="mb-3">
    . . .
    <input name="title" id="txtTitle"
           class="form-control @error('title') is-invalid @enderror"
           value="{{ $bb->title }}">
    @error('title')
    <div class="invalid-feedback">{{ $message }}</div>
    @enderror
</div>
```

Парная директива `@error ... @enderror` шаблонизатора выводит находящийся в ее теле фрагмент содержания только в том случае, если в серверной сессии хранится сообщение об ошибке ввода в элемент управления с указанным именем. Внутри тела директивы можно использовать переменную `message`, хранящую сообщение об ошибке в виде текста.

Мы используем эту директиву, чтобы, во-первых, вывести сообщение об ошибке под полем ввода и, во-вторых, привязать к полю ввода стилевой класс `is-invalid` фреймворка Bootstrap, помечающий элемент управления как содержащий некорректное значение.

4. Добавим туда же код, помещающий в поле ввода ранее занесенное туда значение:

```
<input ... value="{{ old('title', $bb->title) }}">
```

Функция-хелпер `old()` извлекает из серверной сессии значение, занесенное в элемент управления, чье наименование указано в первом параметре.

При первом выводе страницы на экран значение `title` в сессии отсутствует, и функция `old()` вернет значение из второго параметра — изначальное название товара, полученное контроллером из базы данных. Если валидация не увенчалась успехом, страница будет выведена повторно, в этом случае в сессии будет храниться ранее введенное значение `title`, которое и занесется в поле ввода.

5. Внесем аналогичные исправления в код, создающий элементы управления `content` и `price`.
6. Откроем шаблон страницы добавления объявления resources\views\bb-create.blade.php и добавим аналогичный код, выводящий сообщения об ошибках ввода.
7. Добавим в тот же шаблон код, помещающий в элементы управления ранее занесенные туда значения (показаны исправления в коде поля ввода `title` — у остальных элементов правки будут аналогичными):

```
<input ... value="{{ old('title') }}">
```

Указывать второй параметр у функции `old()` здесь не нужно — тогда в случае отсутствия в серверной сессии заданного значения функция вернет `null`, и поле ввода будет выведено пустым.

Войдем на сайт от имени любого пользователя, перейдем на страницу добавления объявления, введем все сведения об объявлении, кроме какого-либо одного (например, названия товара), нажмем кнопку **Добавить** и проверим, выводится ли сообщение об ошибке и заполняются ли остальные элементы управления ранее занесенными туда данными. После этого попробуем ввести название товара длиной более 50 символов и нечисловое значение в качестве цены и посмотрим, что получится.

Пока еще сообщения об ошибках выводятся по-английски. Но мы это обязательно исправим.

По идее, пользователь должен иметь возможность править и удалять лишь «свои» объявления. Однако если мы выполним переход по интернет-адресу формата **/home/<*ключ объявления*>/edit|delete/**, то сможем исправить или удалить объявление с произвольным *ключом* вне зависимости от того, какой пользователь является его автором. Это серьезная брешь в безопасности, и ее следует «заткнуть».

2.5. Разграничение доступа. Посредники, политики и провайдеры

Теория

Разграничение доступа к данным сайта, написанного с применением Laravel, реализуется посредниками и политиками.

- *Посредник* (middleware) — программный модуль, выполняющий предварительную обработку полученного клиентского запроса перед передачей его контроллеру и (или) окончательную обработку ответа после его формирования контроллером и перед отправкой клиенту.

  В частности, посредник `auth` реализует разграничение доступа к сайту. Он проверяет, был ли текущий запрос отправлен пользователем, выполнившим вход на сайт,

и, если это не так, выполняет перенаправление на страницу входа. А посредник `guest`, напротив, проверяет, отправлен ли запрос гостем, в противном случае производя перенаправление на раздел пользователя.

Большинство доступных посредников принадлежат самому фреймворку, но часть непосредственно входит в состав сайта. Они генерируются утилитой artisan при создании проекта и хранятся в папке app\Middleware, так что программист при необходимости вмешаться в обработку запросов и ответов может внести в их код нужные правки.

- *Политика* (policy) — программный модуль, реализующий разграничение доступа к определенной модели согласно заданным правилам.

  В частности, политика может проверять, является ли текущий пользователь автором извлеченного из базы объявления. Если же это не так, будет выведена страница с сообщением об ошибке 403.

  Все политики, входящие в состав сайта, хранятся в папке app\Policies (изначально отсутствующей) и регистрируются в провайдере `AuthServiceProvider`.

- *Провайдер* (provider) — программный модуль, задающий режим работы и некоторые параметры какой-либо из подсистем фреймворка.

  Например, упоминавшийся ранее провайдер `AuthServiceProvider` инициализирует подсистему разграничения доступа и передает ей нужные для работы сведения — в частности, перечень политик.

  Бóльшая часть доступных провайдеров принадлежит самому Laravel. Остальные, в том числе и `AuthServiceProvider`, входят в состав самого сайта, хранятся в папке app\Providers и могут быть исправлены программистом, если он захочет изменить режим работы какой-либо подсистемы фреймворка.

Посредники, политики и провайдеры реализуются в виде классов.

Практика

Сначала удостоверимся, что раздел пользователя, страницы добавления, правки и удаления объявления защищены посредником `auth`, «пускающим» к страницам только пользователей, выполнивших вход. После чего создадим политику `BbPolicy`, которая позволит править и удалять объявления только их автору.

1. Откроем контроллер app\Http\Controllers\HomeController.php и посмотрим на его конструктор:

   ```
   public function __construct() {
       $this->middleware('auth');
   }
   ```

   Метод `middleware()` указывает «пропустить» все запросы, приходящие на действия текущего контроллера, через указанный посредник. Отметим, что вызов этого метода в конструктор класса вставила сама утилита artisan при создании базовых средств разграничения доступа (см. *разд. 2.2*).

   Как видим, все действия контроллера уже защищены от гостей посредником `auth`. Займемся политикой.

2. В командной строке создадим политику `BbPolicy`:

```
php artisan make:policy BbPolicy
```

3. Откроем только что созданную политику app\Policies\BbPolicy.php и посмотрим на ее код (листинг 2.3).

Листинг 2.3. Код «пустой» политики `BbPolicy`

```
namespace App\Policies;

use App\Models\User;
use Illuminate\Auth\Access\HandlesAuthorization;

class BbPolicy {
    use HandlesAuthorization;

    public function __construct() {
    }
}
```

Политика в Laravel — это класс, находящийся в пространстве имен `App\Policies` и использующий трейт `HandlesAuthorization`, содержащий логику, которая допускает пользователя к защищаемой модели лишь после выполнения им входа. Изначально политика содержит только «пустой» конструктор, в принципе ненужный.

4. Добавим в политику `BbPolicy` код, реализующий разграничение доступа:

```
use App\Models\Bb;
class BbPolicy {
    . . .
    public function update(User $user, Bb $bb) {
        return $bb->user->id === $user->id;
    }

    public function destroy(User $user, Bb $bb) {
        return $this->update($user, $bb);
    }
}
```

Логика, реализующая разграничение доступа, записывается в виде набора общедоступных методов класса политики, каждый из которых вызывается при попытке выполнить над записями защищаемой политикой модели определенную операцию (`update()` — правку, `destroy()` — удаление и т. п.). Первым параметром эти методы принимают объект текущего пользователя. Остальные параметры могут быть произвольными — как правило, это объекты моделей, представляющие проверяемые записи. Каждый метод должен возвращать в качестве результата логическую величину: `true` — операция разрешена, `false` — запрещена.

Наши методы получают вторым параметром объект объявления. Метод `update()` сравнивает ключ текущего пользователя с ключом автора объявления, и если они равны (т. е. текущий пользователь является автором объявления), операция правки

разрешается. Метод `destroy()` для выполнения аналогичной проверки просто вызывает метод `update()`.

5. Откроем модуль app\Providers\AuthServiceProvider.php с кодом провайдера `AuthServiceProvider` и добавим код, связывающий модель `Bb` с политикой `BbPolicy`:

```
class AuthServiceProvider extends ServiceProvider {
    . . .
    protected $policies = [
        'App\Models\Bb' => 'App\Policies\BbPolicy',
    ];
    . . .
}
```

   Такие связи записываются в ассоциативном массиве, присвоенном защищенному свойству `policies` класса провайдера. Один элемент массива описывает одну связь между моделью (ее полное имя указывается в качестве ключа элемента) и политикой (полное имя которой станет значением элемента).

6. Откроем список веб-маршрутов routes\web.php и добавим код, указывающий применять политику `BbPolicy` при операциях правки и удаления объявления:

```
Route::get('/home/{bb}/edit', ... )
     ->name('bb.edit')->middleware('can:update,bb');
Route::patch('/home/{bb}', ... )
     ->name('bb.update')->middleware('can:update,bb');
Route::get('/home/{bb}/delete', ... )
     ->name('bb.delete')->middleware('can:destroy,bb');
Route::delete('/home/{bb}', ... )
     ->name('bb.destroy')->middleware('can:destroy,bb');
```

   Объект маршрута, возвращаемый методами `get()`, `post()`, `patch()` и `delete()`, поддерживает знакомый нам метод `middleware()`, задающий посредник, который станет обрабатывать все запросы по текущему маршруту. Мы указали «пропускать» запросы по этим четырем маршрутам через посредник `can`, реализующий разграничение доступа к моделям посредством связанных с ними политик.

   Посредник `can` требует указать два параметра, записываемых после двоеточия через запятую:

   - название выполняемой операции, которое должно совпадать с именем одного из методов, объявленных в политике;
   - имя URL-параметра, содержащего ключ записи. Подсистема внедрения зависимости сама найдет запись по ее ключу и передаст представляющий ее объект указанному методу политики, которая связана с моделью.

   В нашем случае при попытке перейти на страницу правки записи Laravel определит, что с моделью `Bb` связана политика `BbPolicy`, вызовет метод `update()` этой политики и передаст ему объект исправляемого объявления. Метод `update()` проверит, является ли текущий пользователь автором объявления, и разрешит переход на страницу правки или, напротив, выведет сообщение об ошибке 403.

Войдем на сайт от имени любого пользователя и попробуем исправить какое-либо объявление. После чего, набрав интернет-адрес формата **/home/<*ключ объявления*>/edit/**,

попытаемся исправить «чужое» объявление и удостоверимся, что сайт нас к нему не «пускает».

2.6. Получение сведений о текущем пользователе

Осталось сделать так, чтобы гиперссылки на страницы, доступные лишь зарегистрированным пользователям, не показывались гостям, и наоборот. Также неплохо было бы вывести в разделе пользователя его имя — с этого и начнем.

1. Откроем шаблон раздела пользователя resources\views\home.blade.php и вставим код, выводящий заголовок с «адресным» приветствием:

```
@section('content')
<h2>Добро пожаловать, {{ Auth::user()->name }}!</h2>
<p ... ><a ... >Добавить объявление</a></p>
```

   Здесь мы, вызвав метод `user()` фасада `Auth`, получаем объект текущего пользователя и обращаемся к свойству `name` этого объекта, чтобы извлечь имя пользователя.

2. Откроем базовый шаблон resources\views\layouts\app.blade.php и добавим код, выводящий гиперссылки **Регистрация** и **Вход** только гостям, а гиперссылку **Мои объявления** и веб-форму с кнопкой **Выход** — только зарегистрированным пользователям:

```
<a ... >Главная</a>
@guest
<a ... >Регистрация</a>
<a ... >Вход</a>
@endguest
@auth
<a  ... >Мои объявления</a>
<form ... >
    . . .
</form>
@endauth
```

   Парная директива `@quest ... @endguest` шаблонизатора выводит свое содержимое, если вход на сайт не был выполнен, а парная директива `@auth ... @endauth` — если вход, напротив, был выполнен.

Проверим, как все работает.

Страницы регистрации и входа, а также сообщения об ошибках ввода у нас выводятся по-английски. К тому же сообщения об ошибках не очень информативны. Займемся этим.

2.7. Локализация веб-сайта

К счастью, все шаблоны, создаваемые командой `ui:auth`, изначально поддерживают локализацию. Поэтому, чтобы выполнить русскую локализацию этих шаблонов, доста-

точно поместить в состав проекта русские языковые модули и указать в настройках проекта русский в качестве изначального языка сайта.

Локализация — это адаптация сайта, написанного на одном языке (*изначальном*), к другому (*целевому*). Локализация сводится к переводу на целевой язык надписей, выводимых на страницах.

Языковый модуль (или *модуль локализации*) — это программный модуль, который хранит набор выводимых на страницах надписей, переведенных на целевой язык. Laravel позволяет создавать для каждого целевого языка произвольное количество языковых модулей, написанных на языках PHP или JSON.

Чтобы выполнить локализацию сайта, нам понадобится установить в проект две дополнительные библиотеки:

- Laravel Lang — содержащую набор языковых модулей для разных языков (включая русский);
- LaravelLang[1] — добавляющую утилите artisan поддержку новой команды, которая копирует языковые модули для заданного языка из библиотеки Laravel Lang в состав сайта.

Перед установкой новых библиотек лучше остановить отладочный веб-сервер.

1. Установим дополнительную библиотеку Laravel Lang:

   ```
   composer require laravel-lang/lang --dev
   ```

   Ключ `--dev` предписывает Composer пометить устанавливаемую библиотеку как необходимую только на этапе разработки проекта и не требующуюся при его эксплуатации.

2. Установим дополнительную библиотеку LaravelLang:

   ```
   composer require arcanedev/laravel-lang --dev
   ```

   Эта библиотека также требуется только на этапе разработки.

3. Выполним копирование русских языковых модулей из библиотеки Laravel Lang в состав сайта:

   ```
   php artisan trans:publish ru --json
   ```

   После выполнения этой команды в папке lang проекта будет создана папка ru, содержащая набор языковых PHP-модулей. В частности, языковый модуль validation.php, находящийся в этой папке, хранит сообщения об ошибках ввода. Кроме того, непосредственно в папке lang появится языковый JSON-модуль ru.json, содержащий все надписи, которые используются в шаблонах, сгенерированных командой `ui:auth`.

   Обязательно следует указать в настройках проекта русский язык.

4. Откроем модуль config\app.php, содержащий общие настройки проекта, и внесем в него соответствующую правку:

   ```
   return [
       . . .
   ```

[1] Несмотря на похожие названия, это две совершенно разные библиотеки.

```
    'locale' => 'ru',
    . . .
];
```

Запустим отладочный сервер, перейдем на страницу входа и удостоверимся, что все надписи на ней теперь выводятся по-русски (рис. 2.4).

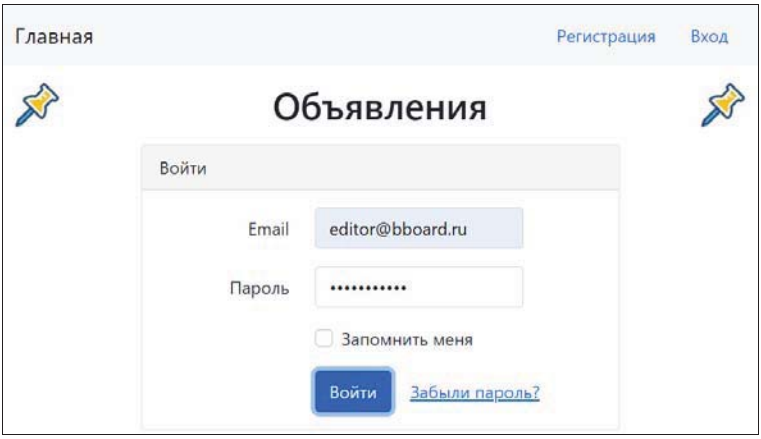


Рис. 2.4. Веб-страница входа после русской локализации

Выполним вход на сайт и попробуем добавить объявление, не указав название товара. Под полем ввода название появится сообщение **Поле title обязательно для заполнения**. `title` — это наименование поля ввода, в которое заносится название товара, и в сообщении об ошибке оно выглядит совершенно не информативно.

Укажем подсистеме локализации Laravel выводить вместо `title` строку **Товар**, вместо `content` (наименование области редактирования, в которое вводится само объявление) — **Содержание объявления**, а вместо `price` (наименование поля для ввода цены) — **Цена**.

5. Откроем русский языковый модуль lang\ru\validation.php, содержащий сообщения об ошибках ввода, и добавим в него код, содержащий эти указания:

```
return [
    . . .
    'attributes' => [
        'title'   => 'Товар',
        'content' => 'Содержание объявления',
        'price'   => 'Цена'
    ]
];
```

И, наконец, сделаем так, чтобы в отсутствии цены выводилось сообщение о недопустимости раздачи товаров даром.

6. Вставим в модуль lang\ru\validation.php следующий код:

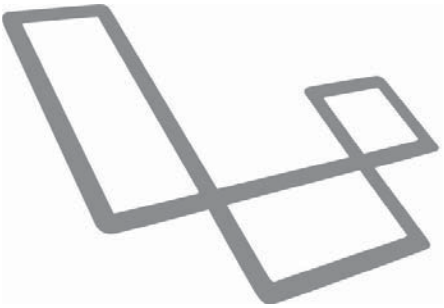
```
return [
    . . .
```

```
        'custom'               => [
            'price' => [
                'required' => 'Раздача товаров даром не допускается',
            ],
        ],
        . . .
    }
```

Добавленный код указывает выводить заданное сообщение об ошибке при нарушении правила валидации `required` (обязательно к заполнению) у поля ввода с наименованием `price` (цена товара).

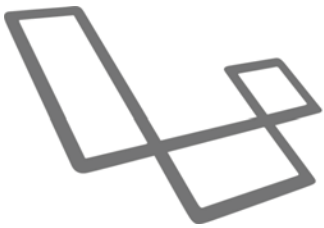
Введем название товара, удалим цену, вновь попытаемся добавить объявление и посмотрим на указанное нами ранее сообщение об ошибке, выведенное на странице.

На этом разработка учебного сайта доски объявлений в основном закончена. Вы можете доработать его самостоятельно в процессе изучения остального материала книги.



ЧАСТЬ II

Базовые инструменты



ГЛАВА 3

Создание, настройка и отладка проекта

3.1. Подготовка к установке

Laravel 9 для работы требует наличия программных платформ следующих (или более новых) версий:

- PHP — 8.0 с расширениями BCMath, Ctype, cURL, DOM, Fileinfo, JSON, Mbstring, OpenSSL, PCRE, PDO, Tokenizer и XML;
- SQLite — 3.8.8;
- MySQL — 5.7;
- MariaDB — 10.2;
- PostgreSQL — 10.0;
- Microsoft SQL Server — 2017.

Отдельный веб-сервер для отладки необязателен — можно использовать отладочный веб-сервер, встроенный в PHP.

Для подготовки программной платформы к разработке сайта на Laravel следует выполнить следующие шаги:

1. Установить платформу PHP (дистрибутив находится на сайте **https://www.php.net/**).
2. Установить программу Composer (**https://getcomposer.org/**).

   В процессе установки потребуется указать путь к файлу php.exe (консольной редакции исполняющей среды PHP) и предписать добавить этот путь в список путей системной переменной `PATH`, установив соответствующий флажок.
3. Установить клиентскую часть используемой серверной СУБД — при необходимости.

3.2. Создание проекта

3.2.1. Создание проекта с помощью Composer

Чтобы создать новый проект, основанный на Laravel 9, следует использовать команду `create-project` программы Composer. Для этого необходимо в командной строке вы-

полнить переход в папку, в которой должна находиться папка создаваемого проекта, и набрать команду следующего формата:

```
composer create-project laravel/laravel <имя проекта> "9.1.*"
```

Папка проекта с указанным *именем* создается в той папке, в которой была отдана команда.

Команда устанавливает библиотеку laravel/laravel, представляющую собой комплект из самого фреймворка Laravel и некоторых дополнительных библиотек (в частности, библиотеки laravel/tinker, реализующей консоль Laravel), версии 9.1, которая описывается в этой книге.

Использование команды `create-project` программы Composer — единственный способ создать проект, основанный на Laravel строго указанной версии.

3.2.2. Создание проекта с помощью Laravel Installer

Создать Laravel-проект также можно с помощью утилиты Laravel Installer. Эта утилита предоставляет дополнительные возможности (в частности, может создать для проекта Git-репозитории — как локальный, так и удаленный).

> ***Внимание!***
>
> К сожалению, посредством утилиты Laravel Installer нельзя создать проект, основанный на строго указанной версии Laravel. Эта утилита всегда устанавливает фреймворк наиболее актуальной на текущий момент версии.

Сначала необходимо установить утилиту Laravel Installer, набрав в командной строке команду:

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

Утилита будет установлена на уровне системы (подробности — в *разд. 1.2*).

Для создания нового Laravel-проекта следует использовать команду формата:

```
laravel new <имя проекта> [--git [--branch=<имя начальной ветви>]] | ↳
[--github[=<ключи консоли Git>] ↳
[--organization=<название организации>]] [--dev] [--force]
```

Поддерживаются следующие полезные ключи:

- `--git` — создать для проекта локальный репозиторий Git[1];
- `--branch` — создать в формируемом локальном репозитории Git изначальную ветвь с заданным *именем*;
- `--github` — создать локальный репозиторий Git и закрытый (private) удаленный репозиторий GitHub[2]. Можно указать командные *ключи*, которые будут переданы утилите GitHub CLI;
- `--organization` — создать удаленный репозиторий GitHub от имени организации с заданным *названием*;

---

[1] Требуется предварительно установить пакет Git (**https://git-scm.com/**).

[2] Требуется предварительно установить утилиту GitHub CLI (**https://cli.github.com/**) и выполнить вход в службу GitHub.

- `--dev` — использовать для проекта самую последнюю версию Laravel, даже если она не является стабильной (релизом);
- `--force` — создать проект, даже если существует папка с именем, совпадающим с указанным *`именем проекта`*.

3.3. Папки и файлы проекта

В этом разделе описаны папки и файлы, составляющие проект и присутствующие в его папке. Вложенность папок и файлов показана отступами. Имена папок набраны прописными буквами, имена файлов — строчными.

- APP — все программные PHP-модули, хранящие код сайта. Распределены по разным папкам, которые мы рассмотрим в последующих главах;
- BOOTSTRAP — модули с инициализационным кодом:
  - CACHE — инициализационные модули, сгенерированные фреймворком и предназначенные для ускорения загрузки библиотечных модулей;
  - app.php — основной инициализационный модуль. В частности, создает объект, представляющий Laravel-сайт;
- CONFIG — все модули с рабочими настройками проекта (будут рассмотрены далее в этой и последующих главах);
- DATABASE — служебные модули, имеющие отношение к базам данных. В этой папке можно сохранить базы данных формата SQLite, используемые сайтом;
  - FACTORIES — фабрики записей (используются при автоматизированном тестировании, которое в этой книге не рассматривается);
  - MIGRATIONS — миграции;
  - SEEDERS — сидеры;
- LANG — языковые файлы, разнесенные по папкам, каждая из которых соответствует одному языку (подробности — в *главе 28*);
- PUBLIC — корневая папка сайта. В нее можно поместить необходимые статические файлы (изображения, таблицы стилей, веб-сценарии и др.):
  - .htaccess — файл конфигурации веб-сервера Apache HTTP Server;
  - favicon.ico — значок сайта;
  - index.php — *единая точка входа* (или *стартовый модуль*). Выполняется при получении любого клиентского запроса, создает объект, представляющий сайт (запуская для этого модуль bootstrap\app.php), запускает обработку запроса, отправляет клиенту сгенерированный ответ и завершает работу сайта;
  - robots.txt — список исключений для поисковых служб;
- RESOURCES — файлы с кодом сайта, написанные на языках, отличных от PHP:
  - CSS — внешние таблицы стилей, написанные на языке CSS;
  - JS — внешние веб-сценарии, подлежащие преобразованию посредством Laravel Mix (будет описан в *главе 17*);
  - VIEWS — шаблоны;

- ROUTES — модули со списками маршрутов;
- STORAGE — файлы, выгруженные на сайт посетителями (изображения, аудио-, видеофайлы, архивы и др.) или созданные программно (например, аватары, сгенерированные на основе выгруженных посетителями изображений), и служебные файлы:
  - APP — файлы, выгруженные на сайт посетителями или созданные программно и не предназначенные для вывода на страницах, — сохраняются непосредственно в этой папке:
    - PUBLIC — файлы, выгруженные на сайт или созданные программно, напротив, выводящиеся на страницах. Обычно в корневой папке сайта (public) создается символическая ссылка `storage`, указывающая на папку storage\app\public (как это сделать, будет описано в *главе 18*);
  - FRAMEWORK — служебные файлы, сгенерированные фреймворком:
    - CACHE\DATA — файлы с кешированными данными (если используется файловый кеш). Подробнее о кешировании данных будет рассказано в *главе 29*;
    - SESSIONS — файлы с серверными сессиями, если в настройках указано хранение сессий в файлах (подробности — в *главе 26*);
    - TESTING — файлы, выгруженные посетителями (сохраняются здесь при работе в тестовом режиме);
    - VIEWS — откомпилированные шаблоны;
  - LOGS — файлы журналов;
- TESTS — тестовые модули.

Автор не описывает в этой книге автоматизированное тестирование...

...поскольку не считает его сколь-нибудь полезным.

- VENDOR — сам фреймворк Laravel и его зависимости — другие библиотеки, используемые им в работе;
- .editorconfig — специфические настройки для текстовых редакторов;
- .env — локальные настройки;
- .env.example — шаблон для генерирования файла .env. Отличается от последнего лишь тем, что не содержит секретного ключа (о нем поговорим позже);
- .gitattributes — конфигурация Git;
- .styleci.yml — конфигурация StyleCI, службы, преобразующей исходный код к заданному стандарту написания;
- artisan — одноименная утилита, служащая для выполнения различных действий над проектом;
- composer.json — конфигурация проекта в формате утилиты Composer;
- composer.lock — список установленных PHP-библиотек с указанием их версий;
- package.json — конфигурация проекта в формате утилиты npm;
- phpuint.xml — конфигурация тестовой подсистемы Laravel;

- readme.md — краткое описание проекта в формате Markdown;
- webpack.mix.js — конфигурация Laravel Mix.

3.4. Настройки проекта

3.4.1. Две разновидности настроек проекта

3.4.1.1. Локальные настройки

Локальные настройки (или *настройки окружения*) описывают некоторые аспекты конфигурации текущей программной платформы: параметры подключения к базе данных, почтовому серверу, параметры кеша, серверных сессий, службы рассылки сообщений и др. Они хранятся в файле .env.

Локальные настройки *не* сохраняются в коммитах Git. Благодаря этому каждый член группы разработчиков, работающих над проектом, может описать в локальных настройках специфические параметры своей платформы. Например, один разработчик может указать в локальных настройках использовать какую-либо систему кеширования (скажем, с целью отладки), а другой — вообще не использовать кеш.

Настройки в файле .env записываются в виде строк формата:

<ИМЯ НАСТРОЙКИ>=<значение настройки>

Имена настроек традиционно набираются в верхнем регистре.

В качестве значения настройки может быть задана:

- строка:

  ```
  APP_NAME=Объявления
  ```

  Если строка содержит пробелы, ее следует заключить в одинарные или двойные кавычки:

  ```
  APP_NAME="Доска объявлений"
  ```

- логическая величина, записанная в виде `true`, `(true)`, `false` или `(false)`:

  ```
  APP_DEBUG=true
  SEND_MAIL=(false)
  ```

- значение `null`, записанное в виде `null` или `(null)`:

  ```
  MAIL_ENCRYPTION=null
  ```

- «пустая» строка, записанная в виде `empty` или `(empty)`:

  ```
  DB_PASSWORD=empty
  ```

- значение другой настройки с указанным *именем*, записанное в формате:

  ```
  "${<ИМЯ НАСТРОЙКИ>}"
  ```

  Например, настройка `MAIL_FROM_NAME` получит значение, указанное у настройки `APP_NAME`:

  ```
  MAIL_FROM_NAME="${APP_NAME}"
  ```

- Если вообще не указать значение у настройки, она в качестве значения получит «пустую» строку:

```
DB_PASSWORD=
```

Пример локальных настроек, задающих параметры доступа к базе данных MySQL:

```
DB_CONNECTION=mysql
DB_HOST=data-server.local
DB_DATABASE=bboard
DB_USERNAME=bboard
DB_PASSWORD=123456789
```

Любую настройку, записанную в файле .env, можно перекрыть, создав переменную среды пользователя или системную переменную с тем же именем (например, если создать переменную `DB_DATABASE` со значением `bboard_new`, Laravel будет использовать базу данных `bboard_new`, а не `bboard`, как записано в файле .env). При этом системные переменные имеют приоритет перед переменными уровня пользователя.

3.4.1.2. Рабочие настройки

Рабочие настройки затрагивают все аспекты функционирования проекта. Именно из рабочих настроек Laravel получает все необходимые ему сведения о сайте.

Рабочие настройки хранятся в 15 модулях, находящихся в папке config. Каждый модуль содержит настройки одной из подсистем фреймворка. Так, модуль database.php хранит настройки подсистемы, обеспечивающей работу с базами данных, модуль session.php — настройки подсистемы серверных сессий, модуль view.php — шаблонизатора и пр. Особняком стоит модуль app.php, содержащий настройки сайта как такового: название, режим работы, язык по умолчанию и др.

Настройки в каждом таком модуле организованы в виде ассоциативного массива, который возвращается из модуля в качестве результата. Отдельный элемент такого массива задает одну из настроек, которая может быть как элементарным значением, так и массивом.

Ряд рабочих настроек получают свои значения из файла .env, в котором хранятся локальные настройки. Таким образом, Laravel при инициализации выполняет объединение локальных и рабочих настроек для удобства программирования.

В качестве примера рассмотрим фрагмент модуля config\database.php:

```
return [
    'default' => env('DB_CONNECTION', 'mysql'),

    'connections' => [
        'sqlite' => [
            'driver' => 'sqlite',
            'url' => env('DATABASE_URL'),
            'database' => env('DB_DATABASE',
                              database_path('database.sqlite')),
            'prefix' => '',
            'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),
        ],
```

```
        . . .
    ],
    . . .
];
```

Элемент `connections` возвращаемого массива содержит список баз данных (более подробно о них мы поговорим очень скоро). А элемент `default` задает базу данных по умолчанию, используемую, если база не была задана явно.

Элемент `default` получает свое значение из локальной настройки `DB_CONNECTION`, записанной в файле .env. Для извлечения оттуда настройки с заданным *именем* используется функция-хелпер `env()`, вызываемая в формате:

`env(<`*имя настройки из файла .env*`>[, <`*значение по умолчанию*`>=null])`

Значение по умолчанию возвращается, если настройка с заданным *именем* в файле .env не найдена.

Также из локальных настроек получают свои значения рабочие настройки `url`, `database` и `foreign_key_constraints`, записанные в массиве `connections.sqlite`.

3.4.2. Настройки проекта по категориям

Рассмотрим наиболее важные настройки самого проекта и используемых им баз данных. Остальные настройки, в том числе задающие режим работы других подсистем фреймворка, будут описаны в последующих главах.

3.4.2.1. Базовые настройки проекта

Базовые настройки проекта записаны в модуле config\app.php. Они включают название проекта, язык по умолчанию, интернет-адрес, по которому опубликован сайт, и пр.:

- `name` — название проекта. Значение берется из локальной настройки `APP_NAME`. По умолчанию: `"Laravel"`;
- `url` — интернет-адрес хоста, на котором работает сайт. Используется утилитой artisan при генерировании модулей. Значение берется из локальной настройки `APP_URL`. По умолчанию: `"http://localhost"`;
- `asset_url` — интернет-адрес хоста, на котором находятся статические файлы сайта, или путь к содержащей их папке без конечного слеша. Используется функцией `asset()` (см. *главу 11*) для формирования интернет-адресов статических файлов.

  Значение настройки берется из локальной настройки `ASSET_URL`, изначально отсутствующей в файле .env. По умолчанию: `null` (указывает, что статические файлы находятся непосредственно в папке public того же хоста, на котором развернут сайт). Пример:

  ```
  // Файл .env
  ASSET_URL=/assets
  ```

  ```
  // Шаблон
  <link href="{{ asset('styles.css') }}" rel="stylesheet">
  <!--
      Результат:
      <link href="/assets/styles.css" rel="stylesheet">
  -->
  ```

- `timezone` — обозначение временно́й зоны по умолчанию в виде строки (по умолчанию: `"UTF"`, т. е. всемирное координированное время):

```
return [
    . . .
    'timezone' => 'Europe/Moscow',
    . . .
];
```

- `locale` — обозначение языка, используемого по умолчанию, если иной язык не задан, в виде строки (по умолчанию `"en"`, т. е. английский). Пример установки русского языка:

```
return [
    . . .
    'locale' => 'ru',
    . . .
];
```

- `fallback_locale` — обозначение языка, используемого, если выбранный посетителем язык сайтом не поддерживается, в виде строки (по умолчанию: `"en"`);
- `faker_locale` — обозначение языка, используемого фабриками записей (применяются при автоматизированном тестировании, которое в этой книге не рассматривается). По умолчанию: `"en_US"` (американский английский).

3.4.2.2. Настройки режима работы веб-сайта

Эти настройки затрагивают особенности функционирования сайта и хранятся в модуле config\app.php.

- `env` — обозначение режима работы сайта в виде строки. Может быть произвольным. Разработчики Laravel рекомендуют указывать следующие режимы:
  - `"local"` — отладочный;
  - `"production"` — эксплуатационный.

  Значение настройки берется из локальной настройки `APP_ENV`. По умолчанию: `"production"`, однако в настройке `APP_ENV` изначально указано `"local"`;
- `debug` — если `true`, при возникновении ошибки будет выводиться страница с подробным описанием проблемы, если `false` — будет выводиться стандартная страница с сообщением об ошибке 503 (внутренняя ошибка сайта). Значение берется из локальной настройки `APP_DEBUG`. По умолчанию: `false`, однако в настройке `APP_DEBUG` изначально указано `true`.

3.4.2.3. Настройки шифрования

Настройки шифрования также хранятся в модуле config\app.php.

- `key` — *секретный ключ*, используемый в операциях шифрования, хеширования и подписывания данных, в виде строки. Значение берется из локальной настройки `APP_KEY`.

Секретный ключ генерируется и записывается в файл .env утилитой artisan при создании нового проекта. При необходимости его можно сгенерировать повторно, отдав команду формата:

```
php artisan key:generate [--force]
```

Если сайт работает в эксплуатационном режиме (настройке env из модуля config\app.php дано значение "production"), утилита artisan спросит, перезаписывать ли старый секретный ключ. Чтобы перезаписать его принудительно, без выдачи запроса, следует указать ключ --force;

- cipher — обозначение алгоритма, применяемого для генерирования секретного ключа, в виде строки. Поддерживаются алгоритмы "AES-128-CBC", "AES-256-CBC" (используется по умолчанию) и "AES-128-GCM" и "AES-256-GCM".

3.4.2.4. Настройки баз данных

Настройки подключения к базам данных, используемым сайтом, хранятся в модуле config\database.php.

- connections — список используемых баз данных (в терминологии Laravel — «подключений»). Их может быть произвольное количество, они могут быть разных форматов, располагаться в разных файлах и на разных хостах. Указывается в виде ассоциативного массива, каждый элемент которого описывает одну базу данных. Ключ элемента этого массива задаст имя базы данных, используемое Laravel.

  Изначально список содержит базы с именами sqlite, mysql, pgsql и sqlsrv форматов соответственно SQLite, MySQL, PostgreSQL и Microsoft SQL Server. Вы можете удалить ненужные базы данных и добавить требуемые.

  Настройки подключения к каждой базе данных также записываются в виде ассоциативного массива (пример которого можно увидеть в *разд. 3.4.1.2*). Эти настройки мы рассмотрим позже;

- default — имя используемой по умолчанию базы данных из указанных в списке connections, если в выражении, обращающемся к данным, база не задана явно. Значение берется из локальной настройки DB_CONNECTION (по умолчанию: "mysql");

- migrations — имя таблицы, создаваемой в базе данных для хранения журнала миграций (подробности будут в *разд. 4.1*). Значение по умолчанию: "migrations", и менять его следует лишь в случае, если в базе уже есть или будет таблица с таким именем.

Настройки самих баз данных, указываемые в элементах массива connections:

- driver — обозначение драйвера PDO, используемого для доступа к базе данных определенного формата, в виде строки: "sqlite" (SQLite), "mysql" (MySQL), "pgsql" (PostgreSQL) и "sqlsrv" (Microsoft SQL Server);

- database — абсолютный путь к файлу базы данных (формата SQLite) или имя базы данных (остальных форматов) в виде строки. Значение берется из локальной настройки DB_DATABASE. По умолчанию:

  - у базы данных sqlite — абсолютный путь к файлу database\database.sqlite, изначально несуществующему;
  - у остальных баз данных — "forge";

- `prefix` — префикс, добавляемый в начале имен всех таблиц, в виде строки. Может пригодиться, если база уже содержит таблицы, чьи имена могут совпасть с именами таблиц с данными сайта. По умолчанию: «пустая» строка.

Следующие настройки указываются лишь у серверных СУБД:

- `host` — интернет-адрес хоста, на котором работает серверная СУБД, в виде строки. Значение берется из локальной настройки `DB_HOST`. По умолчанию: `"127.0.0.1"` или `"localhost"` (локальный хост);
- `port` — номер TCP-порта, через который работает серверная СУБД, в виде строки. Значение берется из локальной настройки `DB_PORT`. По умолчанию:
  - у базы данных `mysql` — `"3306"`;
  - у базы данных `pgsql` — `"5432"`;
  - у базы данных `sqlsrv` — `"1433"`;
- `username` — имя пользователя для подключения к серверной СУБД в виде строки. Значение берется из локальной настройки `DB_USERNAME`. По умолчанию: `"forge"`;
- `password` — пароль для подключения к серверной СУБД в виде строки. Значение берется из локальной настройки `DB_PASSWORD`. По умолчанию: «пустая» строка;
- `charset` — обозначение кодировки баз данных в виде строки. По умолчанию: `"utf8mb4"` (MySQL) или `"utf8"` (остальные);
- `url` — интернет-адрес базы данных, записанный в формате:

  <обозначение драйвера>`://`*<имя пользователя>*`:`*<пароль>*`@`↳
  <интернет-адрес хоста>`[:`*<TCP-порт>*`]/`*<имя базы данных>*`[?`*<параметры>*`]`

  TCP-порт указывается в том случае, если серверная СУБД работает через нестандартный порт. Примеры:

  ```
  mysql://bboard:123456789@data-server.local/bboard?charset=UTF-8
  ```

  Значение берется из локальной настройки `DATABASE_URL`, изначально несуществующей;
- `prefix_indexes` — если `true`, префикс, заданный настройкой `prefix`, будет добавляться в начало имен всех создаваемых таблиц, если `false` — не будет (по умолчанию: `true`).

Далее приведены настройки, специфические для отдельных СУБД:

- `foreign_key_constraints` (только SQLite) — если `true`, ссылочная целостность будет поддерживаться непосредственно на уровне СУБД, если `false` — не будет поддерживаться. Значение берется из локальной настройки `DB_FOREIGN_KEYS`, изначально несуществующей. По умолчанию: `true`;
- `unix_socket` (только MySQL) — обозначение сокета, через который выполняется подключение к серверной СУБД, в виде строки. Значение берется из локальной настройки `DB_SOCKET`. По умолчанию: «пустая» строка;
- `collation` (только MySQL) — обозначение последовательности сортировки записей в виде строки (по умолчанию: `"utf8mb4_unicode_ci"`);

- `strict` (только MySQL) — если `true`, включится «строгий» (strict) режим, если `false` — СУБД будет работать в обычном режиме (по умолчанию: `true`);
- `engine` (только MySQL) — обозначение программного ядра, используемого для работы с базой, в виде строки. По умолчанию: `null` (ядро InnoDB);
- `options` (только MySQL) — ассоциативный массив с дополнительными параметрами подключения. По умолчанию содержит один элемент с ключом `PDO::MYSQL_ATTR_SSL_CA` и значением, которое представляет собой путь к файлу сертификата, извлеченный из локальной настройки `MYSQL_ATTR_SSL_CA`;
- `search_path` (только PostgreSQL) — имя схемы базы данных, с которой будет осуществляться работа, в виде строки (по умолчанию: `"public"`).

> ***Внимание!***
>
> В предыдущих версиях Laravel эта настройка носила имя `schema`. При переносе старого кода на новую версию фреймворка настройку необходимо переименовать.

- `sslmode` (только PostgreSQL) — обозначение режима защищенного подключения в виде строки (по умолчанию: `"prefer"`);
- `encrypt` (только Microsoft SQL Server) — если `"yes"`, обмен данными с СУБД будет производиться по защищенному каналу, если `"no"` — по обычному. Значение берется из локальной настройки `DB_ENCRYPT`, изначально несуществующей. По умолчанию: `"yes"`. Настройка изначально закомментирована;
- `trust_server_sertificate` (только Microsoft SQL Server) — если `true`, сертификат, установленный на сервере, будет считаться доверенным и не станет проверяться, если `false` — не будет считаться доверенным и подвергнется соответствующей проверке. Значение берется из локальной настройки `DB_TRUST_SERVER_CERTIFICATE`, изначально несуществующей. По умолчанию: `false`. Настройка изначально закомментирована.

Если для операций чтения и записи используются разные подключения, специфичные для них параметры записываются в настройках соответственно: `read` и `write`. В рассмотренном далее примере для чтения из базы данных `bboard` используется подключение к хосту `read.data.local` от имени пользователя `r_bboard` с паролем `123456789`, а для записи в ту же базу — подключение к хосту `write.data.local` через нестандартный TCP-порт 6603 от имени пользователя `w_bboard` с паролем `987654321`:

```
'connections' => [
    'mysql' => [
        'driver' => 'mysql',
        'read' => [
            'host' => 'read.data.local',
            'port' => '3306',
            'username' => 'r_bboard',
            'password' => '123456789',
        ],
        'write' => [
            'host' => 'write.data.local',
            'port' => '6603',
```

```
            'username' => 'w_bboard',
            'password' => '987654321',
        ],
        'database' => 'bboard',
        . . .
    ],
],
```

Если параметру `sticky` дать значение `true`, после записи данных через подключение «для записи» чтение также будет выполнено через это же подключение. Это может повысить производительность (разумеется, если пользователь, соединившийся через это подключение, имеет привилегии на чтение данных):

```
'connections' => [
    'mysql' => [
        . . .
        'read' => [ ... ],
        'write' => [ ... ],
        . . .
        'sticky' => true,
    ],
],
```

Если через подключение «для записи» невозможно чтение данных, параметру `sticky` следует дать значение `false` или вообще удалить его.

> ***Полезно знать***
>
> Laravel позволяет использовать несколько баз данных, записанных в настройках проекта. Во фреймворке Django, напротив, для этого требуется дополнительное программирование.

3.4.3. Доступ к настройкам из программного кода

Для извлечения значения рабочей настройки с указанным *путем* применяется функция-хелпер `config()`:

```
config(<путь к настройке>[, <значение по умолчанию>=null])
```

Путь записывается в формате:

```
<имя модуля>.<путь к настройке>
```

Если настройка с заданным *путем* отсутствует, возвращается *значение по умолчанию*. Примеры:

```
// Получаем значение настройки name из модуля config\app.php
$project_name = config('app.name');
// Получаем значение настройки connections.sqlite.database
// из модуля config\database.php
$sqlite_database_path = config('database.connections.sqlite.database');
```

Для программного указания новых значений настроек применяется та же функция `config()`, но в другом формате вызова:

```
config(<ассоциативный массив с задаваемыми настройками>)
```

Ключ элемента заданного *массива* укажет путь к нужной настройке, а значение элемента станет новым значением этой настройки. Пример:

```
// Задаем новое название сайта
config(['app.name' => 'ДО: Доска объявлений']);
```

Выяснить, в каком режиме работает сайт, позволят два следующих метода фасада `Illuminate\Support\Facades\App` (управляющего подсистемой, представляющей сайт как таковой):

- `isLocal()` — возвращает `true`, если сайт работает в отладочном режиме (`local`), и `false` — в противном случае:

```
@if (App::isLocal())
    <p>Сайт работает в отладочном режиме</p>
@endif
```

- `isProduction()` — возвращает `true`, если сайт работает в эксплуатационном режиме (`production`), и `false` — в противном случае.

Также можно использовать метод `environment()` того же фасада, который поддерживает четыре формата вызова:

- `environment()` (без параметров) — возвращает строку с наименованием режима работы:

```
<p>Сайт работает в режиме: {{ App::environment() }}</p>
```

- `environment(<режим>)` — возвращает `true`, если сайт работает в заданном *режиме*, и `false` — в противном случае:

```
@if (App::environment('local'))
    <p>Сайт работает в отладочном режиме (local)</p>
@endif
```

- `environment(<режим 1>, <режим 2>, ... <режим n>)` — возвращает `true`, если сайт работает в одном из указанных *режимов*, и `false` — в противном случае:

```
@if (App::environment('local', 'testing', 'staging'))
    <p>Сайт работает в отладочном или одном из тестовых режимов</p>
@endif
```

- `environment(<массив с режимами>)` — то же самое, что и предыдущий формат вызова:

```
@if (App::environment(['local', 'testing', 'staging']))
    <p>Сайт работает в отладочном или одном из тестовых режимов</p>
@endif
```

3.4.4. Создание своих настроек

Ничто не мешает нам создать свои рабочие настройки, добавив их в один из модулей, хранящихся в папке config. Например, так можно создать в модуле config\app.php настройку `description`, содержащую описание сайта:

```
return [
    . . .
    'description' => 'Электронная доска объявлений',
];
```

А потом — извлечь значение этой настройки:

```
<p>{{ config('app.description') }}</p>
```

Созданная таким образом рабочая настройка может брать значение из локальных настроек (файла .env):

```
// Файл .env
APP_DESC="Электронная доска объявлений"

// Модуль config\app.php
return [
    . . .
    'description' => env('APP_DESC'),
];
```

Также можно создать в папке config новый модуль, предназначенный для хранения вновь добавленных настроек:

```
<?php
// Вновь созданный модуль config\custom.php
return [
    'description' => env('APP_DESC'),
];

// Извлекаем значение настройки custom.description
<p>{{ config('custom.description') }}</p>
```

3.5. Базовые инструменты отладки

3.5.1. Отладочный веб-сервер

Отладочный веб-сервер, встроенный в PHP, запускает команда формата:

```
php artisan serve [--host=<интернет-адрес>] [--port=<TCP-порт>] ↳
[--tries=<количество перебираемых TCP-портов>] [--no-reload]
```

Поддерживаются следующие полезные ключи:

- `--host` — интернет-адрес, с которого будет доступен сайт (по умолчанию: **127.0.0.1** — локальный хост);
- `--port` — номер используемого слушателем порта TCP (по умолчанию: 8000).

  Если указанный порт занят, сервер попытается использовать порт со следующим номером и т. д. Например, если занят порт 8000, будет выполнена попытка использовать порты с номерами 8001, 8002...
- `--tries` — предельное количество TCP-портов, которое следует перебрать в поисках свободного перед выдачей сообщения о невозможности запустить сервер (по умолчанию: 10);
- `--no-reload` — не перезапускать отладочный сервер при изменении файла локальных настроек .env.

3.5.2. Вывод сообщений об ошибках

Если в программном коде сайта допущена ошибка, Laravel выведет стандартную страницу с исчерпывающими сведениями о ней (рис. 3.1).

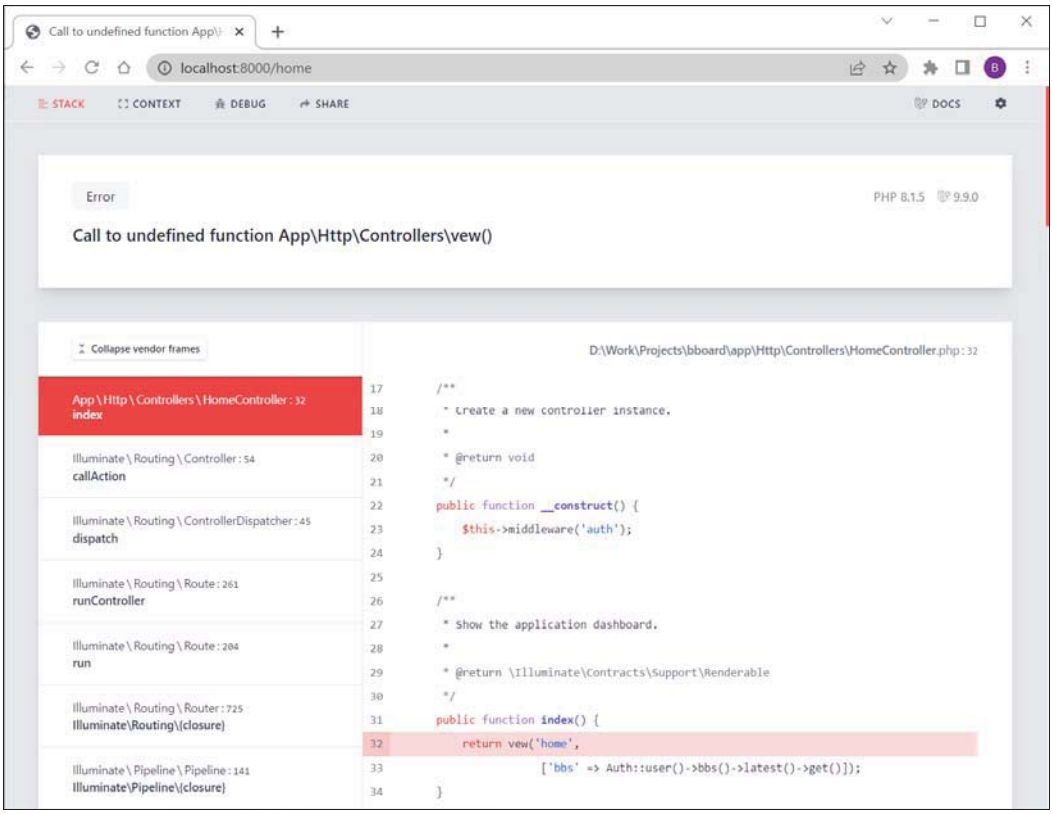


Рис. 3.1. Веб-страница с сообщением об ошибке

В верхней части страницы будет приведено краткое описание возникшей ошибки (на рис. 3.1 — вызов необъявленной функции `vew()`). В правой части будут показаны используемые версии PHP и Laravel.

Ниже отобразятся более подробные сведения об ошибке, разбитые по следующим разделам и подразделам:

- фрагмент исходного кода, в котором возникла ошибка (показан на рис. 3.1). Сведения выводятся в двух списках:
  - в левом — стек вызовов в виде списка модулей. Любой модуль можно выбрать щелчком мыши;
  - в правом — исходный код выбранного в левом списке модуля. Выражение, в котором присутствует ошибка, будет выделено светло-красным фоном;
- **Request** — сведения о клиентском запросе, при обработке которого возникла ошибка:
  - интернет-адрес и содержимое запроса.

Эти, а также некоторые другие описываемые далее сведения, выводятся в спойлере (раскрывающейся панели), раскрыть который можно щелчком на кнопке со стрелкой вниз, расположенной внизу спойлера;

- **Headers** — заголовки запроса в удобочитаемом виде;
- **Body** — данные, переданные в запросе;
- **Session** — содержимое серверной сессии;
- **Coolies** — содержимое cookie;

❒ **App** — сведения о программном модуле, в котором возникла ошибка;

- **Routing** — имена контроллера и действия (**Closure** — если это контроллер-функция), имя указывающего на них маршрута, URL-параметры и их значения, объекты, переданные контроллеру и действию подсистемой внедрения зависимости, и список посредников, связанных с маршрутом;

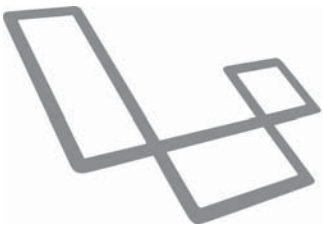
❒ **Context** — сведения о текущем пользователе и самой платформе;

- **User** — имя текущего пользователя, его адрес электронной почты и содержимое записи модели `User`, представляющей текущего пользователя;
- **Version** — версия Laravel, язык сайта, записанный в настройке `app.locale` (локализация сайтов описана в *главе 28*), признак того, кешируются ли настройки сайта (подробности — в *разд. 35.1.8*), значение настройки `app.debug` (см. *разд. 3.4.2.2*), режим работы сайта (см. там же) и версия PHP;

❒ отладочные данные:

- **Queries** — запросы к базе данных, которые были выполнены при попытке открыть запрашиваемую страницу. Над SQL-кодом каждого запроса выводится время, в которое он был выполнен, продолжительность его выполнения в мс и имя базы данных.

В верхней части страницы находится полоса навигации с гиперссылками, указывающими на различные разделы сведений об ошибке: **Stack** (исходный код), **Request** (сведения о клиентском запросе) и **Debug** (отладочные сведения). Находящаяся там же гиперссылка **Share** выводит всплывающее окно, из которого можно опубликовать сведения о возникшей ошибке в Интернете посредством веб-службы Flare (**https://flareapp.io/**), которая специально предназначена для сбора ошибок, возникающих в Laravel-сайтах. Гиперссылка **Docs** ведет на раздел документации «домашнего» сайта Laravel.



ГЛАВА 4

Миграции и сидеры

Миграции предназначены для создания в базах данных необходимых таблиц, полей, индексов и связей, а сидеры — для заполнения таблиц изначальными данными.

4.1. Миграции

Миграция — это программа, вносящая в структуру базы данных заданные изменения. Миграция может создать таблицу со всеми необходимыми полями, индексами и связями, добавить в уже существующую таблицу новое поле или индекс, изменить тип поля, удалить таблицу и пр.

Над любой миграцией можно выполнить одно из следующих действий:

- *применение* — при котором миграция вносит в базу данных заданные изменения.

  Применение миграций выполняется в хронологическом порядке — от созданных ранее к созданным позднее;

- *откат* — при котором миграция возвращает базу данных в изначальное состояние, существовавшее перед применением миграции.

  Откат миграций выполняется в порядке, обратном порядку их применения.

После применения каждой миграции Laravel заносит соответствующую информационную запись в *журнал миграций* — особую таблицу, создаваемую в базе данных по умолчанию. Таблица журнала миграций создается автоматически перед первым применением миграций и получает имя, заданное в рабочей настройке `database.migrations` (по умолчанию: `migrations`).

Обычно PHP-модули с миграциями хранятся в папке database\migrations, однако их можно сохранить и по другому пути (правда, применяя эти миграции, придется указывать путь к ним). Файлы модулей имеют имена формата:

<год>_<№ месяца>_<число>_<часы><минуты><секунды>_<имя миграции>.php

Имя каждого модуля содержит дату и время создания миграции (благодаря чему Laravel без труда сможет выстроить их в хронологическом порядке перед применением), а также произвольно задаваемое *имя*, описывающее назначение миграции.

4.1.1. Создание миграций

Модуль «пустой» миграции создается командой формата:

```
php artisan make:migration <имя миграции> [--table=<имя таблицы>] ↵
[--create=<имя таблицы>] [--path=<путь> [--realpath]] [--fullpath]
```

В *имени миграции* отдельные слова должны разделяться символами подчеркивания.

Если *имя миграции* соответствует одному из следующих форматов:

- `create_<имя таблицы>[_table]` — в миграцию будет добавлен код, создающий таблицу с указанным *именем* и добавляющий в нее ключевое поле, поля отметок создания и правки. Также будет добавлен код, удаляющий эту таблицу при откате миграции;
- `[<произвольный текст>]_to|from|in_<имя таблицы>[_table]` — в миграцию будет добавлен код, открывающий структуру таблицы с указанным *именем* для правки.

В противном случае будет создана полностью «пустая» миграция, и необходимый код придется писать самостоятельно.

Поддерживаются следующие полезные ключи:

- `--table` — принудительно добавляет в миграцию код, открывающий структуру таблицы с указанным в параметре *именем* для правки (даже если указанное в команде *имя миграции* не соответствует упомянутому ранее шаблону);
- `--create` — то же самое, что и `--table`, но дополнительно вставляет код, создающий в таблице ключевое поле, поля отметок создания и правки;
- `--path` — сохраняет модули создаваемых миграций по указанному *пути*. Можно указать как относительный *путь* (от папки проекта), так и абсолютный, добавив ключ `--realpath`;
- `--fullpath` — после создания миграции выводит полный путь к ее модулю.

4.1.2. Класс миграции

Пример «пустой» миграции, сгенерированной утилитой artisan с указанием командного ключа `--create`, можно увидеть в листинге 1.3.

Код, записываемый в модуль миграции, создает и возвращает объект анонимного класса, который и представляет миграцию. Этот класс является производным от класса `Illuminate\Database\Migrations\Migration` и содержит два метода, не принимающих параметров и не возвращающих результаты:

- `up()` — реализует применение миграции (например, создает таблицу);
- `down()` — реализует откат миграции (например, удаляет ранее созданную таблицу).

В предыдущих версиях Laravel миграция представлялась обычным именованным классом. Его объект в коде модуля не создавался и не возвращался. Пример старого кода, объявляющего класс миграции:

```
class CreateBbsTable extends Migration {
    public function up() { ... }

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

Laravel 9 поддерживает миграции, объявленные с применением старого синтаксиса.

По умолчанию все операции, изменяющие структуру базы данных, по возможности выполняются в транзакции. Чтобы указать Laravel не выполнять их в транзакции, следует дать общедоступному свойству `withinTransaction`, унаследованному от суперкласса, значение `false`:

```
class CreateRubricsTable extends Migration {
    public $withinTransaction = false;
    . . .
}
```

Для работы со структурой базы данных применяется фасад `Illuminate\Support\Facades\Schema`, предоставляющий доступ к подсистеме, которая работает со структурой базы данных.

4.1.3. Создание таблиц

Для создания таблицы применяется метод `create()`, вызываемый у фасада `Schema`:

```
create(<имя создаваемой таблицы>,
       <анонимная функция, создающая структуру таблицы>)
```

Анонимная функция должна принимать в качестве единственного параметра объект класса `Illuminate\Database\Schema\Blueprint`, представляющий структуру создаваемой таблицы.

Готовый пример миграции был приведен в *разд. 1.7*.

4.1.3.1. Создание полей

Код, создающий поля новой таблицы, записывается в анонимной функции, переданной вторым параметром методу `create()` фасада `Schema`. Поля создаются особыми методами, вызываемыми у объекта структуры таблицы, который передается анонимной функции в единственном параметре. Большая часть этих методов и типов создаваемых ими полей приведены далее (остальные будут описаны позже):

- `string(<имя поля>[, <предельная длина строки в символах>=null])` — *строковое поле* типа `VARCHAR`, хранящее строку ограниченной длины. Если *предельная длина* не задана, она будет взята из общедоступного статического свойства `defaultStringLength` класса `Illuminate\Database\Schema\Builder` (по умолчанию: 255).

  Можно указать другую длину строки по умолчанию: с помощью метода `defaultStringLength(<длина строки по умолчанию>)` фасада `Schema`. Вызов этого метода следует записать в теле метода `boot()` провайдера `App\Providers\AppServiceProvider`. Пример:

  ```
  use Illuminate\Support\Facades\Schema;
  class AppServiceProvider extends ServiceProvider {
      . . .
      public function boot() {
          . . .
          Schema::defaultStringLength(127);
      }
  }
  ```

- `char(<имя поля>[, <предельная длина строки в символах>=null])` — то же самое, что и `string()`, но создается поле типа `CHAR`, хранящее строку фиксированной длины. Слишком короткие строки будут дополняться до нужной длины пробелами справа;
- `text(<имя поля>)` — *текстовое поле*, хранящее строку произвольной длины, но не более 65 536 символов;
- `mediumText(<имя поля>)` — текстовое поле с предельным объемом 16 777 216 символов;
- `longText(<имя поля>)` — текстовое поле с предельным объемом 4 294 967 296 символов;
- `integer()` — знаковое целочисленное поле размером 4 байта:

```
integer(<имя поля>[, <автоинкрементное?>=false[,
                     <беззнаковое?>=false]])
```

- `unsignedInteger()` — беззнаковое целочисленное поле размером 4 байта:

```
unsignedInteger(<имя поля>[, <автоинкрементное?>=false])
```

> ***Беззнаковые числовые поля поддерживаются лишь MySQL***
>
> Остальные форматы баз данных их не поддерживают, и при попытке создать в них беззнаковое поле будет создано обычное поле.

- `bigInteger()` — знаковое целочисленное поле размером 8 байтов. Формат вызова такой же, как у метода `integer()`;
- `unsignedBigInteger()` — беззнаковое целочисленное поле размером 8 байтов. Формат вызова такой же, как у метода `unsignedInteger()`;
- `mediumInteger()` — знаковое целочисленное поле размером 3 байта. Формат вызова такой же, как у метода `integer()`;
- `unsignedMediumInteger()` — беззнаковое целочисленное поле размером 3 байта. Формат вызова такой же, как у метода `unsignedInteger()`;
- `smallInteger()` — знаковое целочисленное поле размером 2 байта. Формат вызова такой же, как у метода `integer()`;
- `unsignedSmallInteger()` — беззнаковое целочисленное поле размером 2 байта. Формат вызова такой же, как у метода `unsignedInteger()`;
- `tinyInteger()` — знаковое целочисленное поле размером 1 байт. Формат вызова такой же, как у метода `integer()`;
- `unsignedTinyInteger()` — беззнаковое целочисленное поле размером 1 байт. Формат вызова такой же, как у метода `unsignedInteger()`;
- `float()` — знаковое вещественное число обычной точности (размером 4 байта):

```
float(<имя поля>[, <общее количество цифр>=8[,
      <количество цифр после запятой>=2[, <беззнаковое?>=false]]])
```

  Не все СУБД поддерживают указание *общего количества цифр* и *количества цифр после запятой*;

- `unsignedFloat()` — беззнаковое вещественное число обычной точности (размером 4 байта):

```
unsignedFloat(<имя поля>[, <общее количество цифр>=8[,
                          <количество цифр после запятой>=2]])
```

Не все СУБД поддерживают указание *общего количества цифр* и *количества цифр после запятой*;

- `double()` — знаковое вещественное число двойной точности (размером 8 байтов):

```
double(<имя поля>[, <общее количество цифр>=null[,
       <количество цифр после запятой>=null[, <беззнаковое?>=false]]])
```

Если *общее количество цифр* и *количество цифр после запятой* не указаны или равны `null`, в поле можно хранить вещественные числа с произвольным количеством цифр. Следует отметить, что не все СУБД поддерживают указание этих параметров;

- `unsignedDouble()` — беззнаковое вещественное число двойной точности (размером 8 байтов):

```
unsignedDouble(<имя поля>[, <общее количество цифр>=null[,
                           <количество цифр после запятой>=null]])
```

Если *общее количество цифр* и *количество цифр после запятой* не указаны или равны `null`, в поле можно хранить вещественные числа с произвольным количеством цифр. Не все СУБД поддерживают указание этих параметров;

- `decimal()` — вещественное число высокой точности. Может использоваться для хранения денежных сумм. Формат вызова такой же, как у метода `float()`. Пример:

```
$table->decimal('price', 10, 2, true);
```

- `unsignedDecimal()` — беззнаковое вещественное число высокой точности. Формат вызова такой же, как у метода `unsignedFloat()`. Пример:

```
$table->unsignedDecimal('price', 10, 2);
```

- `dateTime(<имя поля>[, <точность>=0])` — временна́я отметка (поле типа `DATETIME`). Параметр *точность* указывает количество цифр после запятой, отводимых для хранения долей секунд (например, если указать точность, равную 3, можно будет хранить временны́е отметки с точностью до миллисекунды);
- `dateTimeTz(<имя поля>[, <точность>=0])` — то же самое, что и `dateTime()`, но с учетом временно́й зоны;
- `timestamp(<имя поля>[, <точность>=0])` — то же самое, что и `dateTime()`, но создается поле типа `TIMESTAMP`;
- `timestampTz(<имя поля>[, <точность>=0])` — то же самое, что и `timestamp()`, но с учетом временно́й зоны;
- `timestamps([<точность>=0])` — создает необязательные для заполнения поля типа `TIMESTAMP` для хранения отметок создания и правки с именами `created_at` и `updated_at` соответственно. Параметр *точность* указывает количество цифр после запятой, отводимых для хранения долей секунд;
- `nullableTimestamps([<точность>=0])` — то же, что и `timestamps()`;
- `timestampsTz([<точность>=0])` — то же самое, что и `timestamps()`, но с учетом временно́й зоны;

- `date(<имя поля>)` — дата;
- `time(<имя поля>[, <точность>=0])` — время. Параметр *точность* указывает количество цифр после запятой, отводимых для хранения долей секунд;
- `timeTz(<имя поля>[, <точность>=0])` — то же самое, что и `time()`, но с учетом временно́й зоны;
- `year(<имя поля>)` — год;
- `boolean(<имя поля>)` — логическая величина;
- `bigIncrements(<имя поля>)` — ключевое автоинкрементное беззнаковое целочисленное поле размером 8 байтов. Также автоматически создает ключевой индекс по этому полю;
- `id([<имя поля>='id'])` — то же, что и `bigIncrements()`. По умолчанию создает поле с именем `id`;
- `increments(<имя поля>)` — ключевое автоинкрементное беззнаковое целочисленное поле размером 4 байта;
- `mediumIncrements(<имя поля>)` — ключевое автоинкрементное беззнаковое целочисленное поле размером 3 байта;
- `smallIncrements(<имя поля>)` — ключевое автоинкрементное беззнаковое целочисленное поле размером 2 байта;
- `tinyIncrements(<имя поля>)` — ключевое автоинкрементное беззнаковое целочисленное поле размером 1 байт;
- `uuid(<имя поля>)` — универсальный уникальный идентификатор (UUID). Может использоваться для идентификации записи вместо уникального номера;
- `rememberToken()` — необязательное для заполнения строковое поле `remember_token` длиной 100 символов, в котором будет храниться электронный жетон для запоминания пользователя (подробнее об этом будет рассказано в *главе 13*);
- `enum(<имя поля>, <массив допустимых значений>)` — любое строковое значение из заданного *массива* (*поле перечисления*):

```
$table->enum('type', ['Продажа', 'Купля', 'Обмен']);
```

- `set(<имя поля>, <массив допустимых значений>)` — произвольное количество любых строковых значений из заданного *массива* (*поле набора*):

```
$table->set('others', ['Срочно!', 'Возможен торг', 'Возможен обмен']);
```

- `json(<имя поля>)` — данные в формате JSON;
- `jsonb(<имя поля>)` — данные в формате JSONB;
- `binary(<имя поля>)` — двоичные данные произвольной длины (`BLOB`);
- `ipAddress(<имя поля>)` — IP-адрес;
- `macAddress(<имя поля>)` — MAC-адрес;
- `geometry(<имя поля>)` — описание геометрической фигуры;
- `point(<имя поля>)` — описание геометрической точки;

- `lineString(<имя поля>)` — описание геометрической линии;
- `polygon(<имя поля>)` — описание геометрического полигона;
- `geometryCollection(<имя поля>)` — набор описаний геометрических фигур;
- `multiPoint(<имя поля>)` — набор описаний геометрических точек;
- `multiLineString(<имя поля>)` — набор описаний геометрических линий;
- `multiPolygon(<имя поля>)` — набор описаний геометрических полигонов.

Все эти методы возвращают объект класса `\Illuminate\Database\Schema\ColumnDefinition`, представляющий описание созданного поля.

4.1.3.2. Реализация «мягкого» удаления записей в таблицах

При *«мягком»* удалении запись не удаляется из таблицы, а всего лишь помечается занесением текущей временно́й отметки в особое поле (*отметки удаления*). Чтобы восстановить «мягко» удаленную запись, следует очистить это поле. Все эти операции выполняются самим фреймворком.

При выборке данных записи, подвергшиеся «мягкому» удалению, не извлекаются (однако при необходимости их все же можно извлечь вместе с записями, не подвергшимися «мягкому» удалению, — как это сделать, будет рассказано в *главе 7*).

Создание в таблице поля отметки удаления выполняется вызовом одного из следующих методов класса `Blueprint`:

- `softDeletes([<имя поля>='deleted_at'[, <точность>=0]])` — отметка удаления (тип `TIMESTAMP`). По умолчанию создаваемое поле получит имя `deleted_at`. Параметр *точность* указывает количество цифр после запятой, отводимых для хранения долей секунд;
- `softDeletesTz([<имя поля>='deleted_at'[, <точность>=0]])` — то же самое, что и `softDeletes()`, только с учетом временно́й зоны.

Пример:

```
public function up() {
    Schema::create('bbs', function (Blueprint $table) {
        . . .
        $table->softDeletes();
    });
}
```

4.1.3.3. Указание дополнительных параметров полей

Для указания дополнительных параметров полей применяются методы, приведенные далее. Они вызываются у объекта, который возвращается методами, описанными в *разд. 4.1.3.1* и *4.1.3.2*, и представляет создаваемое поле:

- `default(<значение по умолчанию>)` — задает для текущего поля указанное *значение по умолчанию*:

```
// Указываем значение цены по умолчанию: 0
$table->decimal('price', 10, 2)->default(0);
```

Если в качестве значения по умолчанию требуется использовать выражение языка SQL, следует оформить его в виде объекта класса `Illuminate\Database\Query\Expression`. Конструктору этого класса надо передать строку с нужным SQL-выражением. Пример записи в поле `random` в качестве значения по умолчанию случайного числа, вычисленного SQL-функцией `RAND()`:

```
use Illuminate\Database\Query\Expression;
return new class extends Migration {
    public function up() {
        Schema::create('sometable', function (Blueprint $table) {
            . . .
            $table->float('random')
                  ->default(new Expression('RAND()'));
        });
    }
}
```

- `nullable([`*<может хранить null?>*`=true])` — превращает текущее поле в необязательное (если с параметром передано значение `true`) или, наоборот, обязательное (если передано `false`) для заполнения:

```
// Помечаем поле desc как необязательное для заполнения
$table->text('desc')->nullable();
```

- `useCurrent()` — задает для текущего поля временно́й отметки в качестве значения по умолчанию текущие дату и время;
- `useCurrentOnUpdate()` — указывает занести в текущее поле временно́й отметки текущие дату и время при каждом сохранении записи;
- `autoincrement()` — превращает текущее поле (должно быть целочисленным) в автоинкрементное;
- `from(<`*начальное значение*`>)` (только MySQL и PostgreSQL) — указывает у текущего автоинкрементного поля заданное *начальное значение*;
- `storedAs(<`*SQL-выражение*`>)` (только MySQL и PostgreSQL) — превращает текущее поле в хранимое вычисляемое, чье значение рассчитывается на основе заданного *SQL-выражения* и сохраняется в таблице:

```
// Значение поля total будет представлять собой произведение
// значений полей price и count
$table->decimal('total', 10, 2)->storedAs('`price` * `count`');
```

- `comment(<`*комментарий*`>)` (только MySQL и PostgreSQL) — добавляет текущему полю произвольный строковый *комментарий*;
- `virtualAs(<`*SQL-выражение*`>)` (только MySQL) — то же самое, что и `virtualAs()`, только превращает текущее поле в простое вычисляемое (чье значение *не* сохраняется в таблице);
- `unsigned()` (только MySQL) — превращает текущее целочисленное поле в беззнаковое;
- `invisible()` (только MySQL) — превращает текущее поле в невидимое, и его значение не будет извлекаться при выполнении SQL-команды `SELECT`;

- `charset(<обозначение кодировки>)` (только MySQL) — указывает у текущего поля текстовую кодировку с заданным *обозначением*;
- `collation(<обозначение последовательности сортировки>)` (только MySQL) — указывает у текущего поля последовательность сортировки с заданным *обозначением*;
- `first()` (только MySQL) — помещает текущее поле в самое начало таблицы;
- `after(<имя поля>)` (только MySQL) — помещает текущее поле после поля с заданным *именем*;
- `generatedAs([<параметры>=null])` (только PostgreSQL) — превращает текущее целочисленное поле в поле идентификации, заполняемое только в том случае, если значение не было занесено в него явно:

```
// Создаем ключевое поле
$table->unsignedBigInteger('id')->generatedAs();

// Дополнительно указываем: начать нумерацию записей с 10
// и увеличивать следующий номер на 5
$table->unsignedBigInteger('id')
      ->generatedAs('start with 10 increment by 5');
```

- `always()` (только PostgreSQL) — превращает текущее поле идентификации в заполняемое принудительно:

```
$table->unsignedBigInteger('id')->generatedAs()->always();
```

- `isGeometry()` (только PostgreSQL) — указывает у текущего поля с описанием геометрической фигуры тип `GEOMETRY` вместо используемого для таких полей по умолчанию типа `GEOGRAPHY`.

4.1.3.4. Создание индексов

Далее приведены методы, создающие в таблице индексы разных типов:

- `index()` — обычный индекс. Поддерживает три формата вызова:
  - `index([<имя индекса>=null[, <алгоритм>=null]])`

    Вызывается у объекта, представляющего создаваемое поле:

    ```
    $table->unsignedTinyInteger('order')->index();
    $table->string('name', 40)->index('idx_name', 'hash');
    ```

    Создание индекса с применением разных алгоритмов поддерживают не все СУБД. Если *алгоритм* не указан, индекс будет создан с применением алгоритма по умолчанию, зависящего от конкретной СУБД;

  - `index(<имя индексируемого поля>[, <имя индекса>=null[, <алгоритм>=null]])`

    Вызывается у объекта, представляющего структуру создаваемой таблицы:

    ```
    $table->string('name', 40);
    $table->index('name');
    ```

  - `index(<массив с именами индексируемых полей>[, <имя индекса>=null[, <алгоритм>=null]])`

Создает составной индекс сразу по нескольким полям:

```
$table->unsignedTinyInteger('order');
$table->string('name', 40);
$table->index(['name', 'order']);
```

- `unigue()` — уникальный индекс. Формат вызова такой же, как у метода `index()`;
- `primary()` — ключевой индекс. Формат вызова такой же, как у метода `index()`.

  Этот метод следует вызывать у ключевых полей, не являющихся автоинкрементными (у автоинкрементных полей ключевой индекс создается автоматически);
- `fullText()` (только MySQL и PostgreSQL) — полнотекстовый индекс. Формат вызова такой же, как у метода `index()`.

  Базы данных PostgreSQL поддерживают указание языка, для которого будет создан полнотекстовый индекс. Для этого применяется метод `language(<обозначение языка>)`, который вызывается у объекта, возвращенного методом `fullText()`. Пример:

  ```
  $table->fullText('content')->language('russian');
  ```
- `spatialIndex()` (только MySQL) — пространственный (spatial) индекс. Формат вызова такой же, как у метода `index()`, только *алгоритм* не указывается;
- `rawIndex(<SQL-выражение>, <имя индекса>)` — индекс на основе *SQL-выражения*, которое должно быть задано в виде объекта класса `Expression`:

```
// Создаем индекс на основе значения поля name,
// приведенного к верхнему регистру
use \Illuminate\Database\Query\Expression;
return new class extends Migration {
    public function up() {
        Schema::create('sometable', function (Blueprint $table) {
            . . .
            $table->string('name', 40);
            $table->rawIndex(new Expression('upper(name)'),
                             'idx_name');
        });
    }
}
```

Если в вызове любого из приведенных методов, кроме `rawIndex()`, не указано *имя индекса*, созданный индекс получит имя формата:

<имя таблицы>_<имя поля>_<тип индекса: index, unique или primary>

4.1.3.5. Создание полей внешнего ключа

Поле внешнего ключа участвует в установлении межтабличной связи. Оно хранит ключ связываемой записи первичной таблицы и, таким образом, создается во вторичной таблице одним из двух способов.

Первый способ — простой — реализуется в два этапа:

1. Создание поля внешнего ключа вместе с индексом внешнего ключа — вызовом у объекта структуры создаваемой таблицы метода `foreignIdFor()`:

```
foreignIdFor(<имя класса связываемой модели первичной таблицы>[,
            <имя создаваемого поля>=null])
```

Если *имя создаваемого поля* не указано, будет создано поле с именем формата:

```
<имя класса первичной модели>_<имя ключевого поля первичной модели>
```

Имя класса первичной модели будет набрано в нижнем регистре, а отдельные слова в нем — разделены символом подчеркивания (стиль именования *snake_case*). Например, поле внешнего ключа для связи с моделью `Rubric`, имеющей ключевое поле `id`, получит имя `rubric_id`.

Метод `foreignIdFor()` создает поле внешнего ключа того же типа, к которому принадлежит ключевое поле связываемой первичной таблицы. Так, если первичная таблица имеет ключевое поле типа UUID, будет создано поле внешнего ключа того же типа.

Также можно использовать один из следующих методов, создающих поля внешнего ключа определенных типов:

- `foreignId(<имя поля>)` — создает беззнаковое целочисленное поле размером 8 байтов;
- `foreignUuid(<имя поля>)` — создает поле типа UUID.

2. Создание собственно связи — вызовом метода `constrained()` у созданного поля внешнего ключа:

```
constrained([<имя первичной таблицы в единственном числе>=null[,
            <имя ключевого поля первичной таблицы>='id']])
```

Если имя поля внешнего ключа соответствует формату:

```
<имя связываемой первичной таблицы в единственном числе>_id
```

а ключевое поле связываемой первичной таблицы называется `id`, параметры в вызове метода `constrained()` можно не указывать — фреймворк извлечет все необходимые сведения из имени поля внешнего ключа.

К сожалению, первым способом нельзя создать поле внешнего ключа, имеющее какой-либо иной тип (например, строковое или беззнаковое целое 4-байтовое).

Примеры:

```
// Создаем поле внешнего ключа для связи с первичной таблицей,
// представляемой моделью Rubric. Поле получит имя rubric_id.
use App\Models\Rubric;
. . .
$table->foreignIdFor(Rubric::class)->constrained();

// Другой способ создать это поле
$table->foreignId('rubric_id')->constrained();

// Поле внешнего ключа user свяжет текущую вторичную таблицу
// с первичной таблицей userlist, имеющей ключевое поле num
$table->foreignId('user')->constrained('userlist', 'num');
```

Второй — сложный — способ реализуется в четыре этапа:

1. Создание поля внешнего ключа типа, совпадающего с типом ключевого поля первичной таблицы, — вызовом соответствующего метода у класса структуры создаваемой таблицы. Так, если требуется создать беззнаковое целочисленное 8-байтовое поле, следует вызвать метод `unsignedBigInteger()`.
2. Создание индекса внешнего ключа на основе этого поля — вызовом метода `foreign()` у объекта структуры таблицы:

   `foreign(<имя созданного поля>[, <имя индекса>=null])`

   Если *имя индекса* не указано, созданный индекс получит имя формата:

   `<имя текущей таблицы>_<имя поля>_foreign`

   Метод возвращает объект, представляющий созданный индекс внешнего ключа.
3. Указание у созданного внешнего ключа ключевого поля связываемой первичной таблицы — вызовом у него метода `references()`:

   `references(<имя ключевого поля связываемой первичной таблицы>)`
4. Указание у созданного внешнего ключа самой связываемой первичной таблицы — вызовом у него метода `on()`:

   `on(<имя связываемой первичной таблицы>)`

Второй способ позволяет создать поле внешнего ключа любого типа.

Пример:

```
// Поле внешнего ключа rubric_id свяжет текущую вторичную таблицу
// с первичной таблицей rubrics, имеющей ключевое поле id
$table->unsignedBigInteger('rubric_id');
$table->foreign('rubric_id')->references('id')->on('rubrics');
```

Указать, какую операцию следует выполнять с записями вторичной таблицы при изменении значения поля в связанной записи первичной таблицы или при удалении этой записи, можно вызовом у объекта внешнего ключа следующих методов:

- `onDelete(<обозначение операции>)` — операция, выполняемая при удалении записи первичной таблицы. Обозначение операции задается в формате СУБД. Пример:

  ```
  // Указываем при удалении записи первичной таблицы выполнять
  // каскадное удаление связанных записей вторичной таблицы
  $table->foreignId('rubric_id')->constrained()->onDelete('cascade');
  ```
- `cascadeOnDelete()` — то же самое, что и `onDelete('cascade')`;
- `restrictOnDelete()` — то же самое, что и `onDelete('restrict')`;
- `nullOnDelete()` — при удалении записи первичной таблицы — занести в поле внешнего ключа связанных записей вторичной таблицы значение `null`;
- `onUpdate(<обозначение операции>)` — операция, выполняемая при изменении значения ключевого поля у записи первичной таблицы;
- `cascadeOnUpdate()` — то же самое, что и `onUpdate('cascade')`.

На основе поля внешнего ключа создается индекс, называемый *внешним ключом* и имеющий имя формата:

```
<имя вторичной таблицы>_<имя поля внешнего ключа>_↩
<имя ключевого поля первичной таблицы>_foreign
```

Например, на основе поля внешнего ключа `rubric_id` таблицы `bbs` будет создан внешний ключ `bbs_rubric_id_foreign`. Впоследствии при необходимости можно удалить внешний ключ, указав его имя.

4.1.3.6. Задание дополнительных параметров таблиц

Дополнительные параметры таблиц можно указать с помощью следующих свойств объекта, представляющего структуру создаваемой таблицы:

- `charset` (только MySQL) — текстовая кодировка у всей таблицы;
- `collation` (только MySQL) — последовательность сортировки записей у всей таблицы;
- `engine` (только MySQL) — программное ядро, используемое для работы с таблицей.

Метод `temporary()` структуры таблицы указывает создать временную таблицу (не поддерживается Microsoft SQL Server).

Пример:

```
Schema::create('rubrics', function (Blueprint $table) {
    $table->charset = 'cp1251';
    $table->collation = 'cp1251_ukrainian_ci';
    $table->temporary();
    $table->id();
    . . .
});
```

4.1.4. Правка и удаление таблиц

4.1.4.1. Правка и удаление полей

Для правки полей (а также индексов, о чем речь пойдет позже) таблицы применяется метод `table()`. Он также вызывается у фасада `Schema` и аналогичен методу `create()`, рассмотренному в *разд. 4.1.3*. Пример:

```
Schema::table('rubrics', function (Blueprint $table) {
    // Код, правящий структуру таблицы, записывается здесь
});
```

> ***ПЕРЕД ПРАВКОЙ ПОЛЕЙ...***
>
> ...необходимо установить дополнительную библиотеку `doctrine/dbal`, отдав команду:
>
> ```
> composer require doctrine/dbal
> ```
>
> Если используется база данных формата Microsoft SQL Server, установку библиотеки следует выполнять отдачей команды:
>
> ```
> composer require doctrine/dbal:^3.0
> ```

- Добавление поля — выполняется способами, описанными в *разд. 4.1.3*.
- Правка поля — выполняется в два этапа:
  - задание новых параметров поля — вызовом необходимого метода из числа описанных в *разд. 4.1.3.1*, в нем указывается имя исправляемого поля и его новые параметры. Также можно задать у поля дополнительные параметры — вызовом методов из *разд. 4.1.3.3*, и создать индекс — вызовом методов из *разд. 4.1.3.4*;
  - указание исправить поле — вызовом сцепляемого метода `change()` у объекта поля.

  Пример:

```
// Добавляем в таблицу rubrics поле description, увеличиваем длину
// поля name до 50 символов и создаем на его основе уникальный индекс
public function up() {
    Schema::table('rubrics', function (Blueprint $table) {
        $table->text('description');
        $table->string('name', 50)->unique()->change();
    });
}
```

- Переименование поля — вызовом метода `renameColumn()` структуры таблицы:

```
renameColumn(<старое имя поля>, <новое имя поля>)
```

  Пример:

```
// Переименовываем поле description в desc
Schema::table('rubrics', function (Blueprint $table) {
    $table->renameColumn('description', 'desc');
});
```

> ***ПЕРЕИМЕНОВАНИЕ ПОЛЕЙ ПЕРЕЧИСЛЕНИЯ…***
>
> …в настоящее время не поддерживается.

- Удаление поля — вызовом метода `dropColumn()` структуры таблицы. Метод поддерживает два формата:

```
dropColumn(<имя поля 1>, <имя поля 2>, ... <имя поля b>)
dropColumn(<массив имен полей>)
```

  Пример удаления из таблицы `rubrics` поля `description`:

```
Schema::table('rubrics', function (Blueprint $table) {
    $table->dropColumn('desc');
});
```

  Удалить служебное поле также можно вызовом одного из следующих методов:

  - `dropTimestamps()` — удаляет поля отметок создания и правки;
  - `dropTimestampsTz()` — то же самое, что и `dropTimestamps()`;
  - `dropRememberToken()` — удаляет поле `remember_token` с электронным жетоном для запоминания пользователя;

- `dropSoftDeletes()` — удаляет поле отметки удаления;
- `dropSoftDeletesTz()` — то же самое, что и `dropSoftDeletes()`.

Правка или удаление нескольких полей одновременно…
…в базах данных SQLite в настоящее время не поддерживается.

4.1.4.2. Переименование и удаление индексов

- Переименование индекса — выполняется вызовом метода `renameIndex()` у структуры таблицы:

```
renameIndex(<старое имя индекса>, <новое имя индекса>)
```

Пример:

```
// Переименовываем уникальный индекс, созданный ранее по полю name и
// получивший имя по умолчанию rubrics_name_unique, в udx_name
Schema::table('rubrics', function (Blueprint $table) {
    $table->renameIndex('rubrics_name_unique', 'udx_name');
});
```

- Удаление индекса — вызовом одного из следующих методов у структуры таблицы:
  - `dropIndex(<имя индекса>)` — удаляет обычный индекс;
  - `dropUnique(<имя индекса>)` — удаляет уникальный индекс;
  - `dropPrimary(<имя индекса>)` — удаляет ключевой индекс;
  - `dropFullText(<имя индекса>)` — удаляет полнотекстовый индекс;
  - `dropSpatial(<имя индекса>)` — удаляет пространственный индекс.

Пример удаления уникального индекса `udx_name`:

```
$table->dropUnique('udx_name');
```

Если индекс имеет имя по умолчанию, сгенерированное самим фреймворком, вместо его *имени* можно указать массив с именами полей, на основе которых он создан. Пример удаления индекса, созданного на основе полей `title` и `price`, при условии, что он имеет имя по умолчанию:

```
$table->dropIndex(['title', 'price']);
```

4.1.4.3. Удаление полей внешнего ключа и управление соблюдением ссылочной целостности

Удаление поля внешнего ключа вместе с индексом выполняется вызовом метода `dropForeign()` у структуры таблицы:

```
dropForeign(<имя внешнего ключа>|<массив с именем поля внешнего ключа>)
```

Передаваемый массив должен содержать единственный элемент — имя поля внешнего ключа. Удаляем связь текущей вторичной таблицы `bbs` с первичной таблицей `rubrics` двумя способами:

```
$table->dropForeign('bbs_rubric_id_foreign');
$table->dropForeign(['rubric_id']);
```

Перед изменением структуры связанной таблицы, возможно, потребуется временно запретить соблюдение ссылочной целостности, а потом вновь разрешить его. Это выполняется вызовом у фасада `Schema` методов `disableForeignKeyConstraints()` и `enableForeignKeyConstraints()` соответственно:

```
Schema::disableForeignKeyConstraints();
Schema::table('bbs', function (Blueprint $table) {
    // Правим структуру таблицы
});
Schema::enableForeignKeyConstraints();
```

4.1.4.4. Переименование и удаление таблиц

- Переименование таблицы — выполняется вызовом метода `rename()` у фасада `Schema`:

  `rename(<старое имя таблицы>, <новое имя таблицы>)`

  Пример:

  ```
  Schema::rename('rubrics', 'rubric_list');
  ```

- Удаление таблицы — вызовом одного из двух методов у фасада `Schema`:

  - `drop(<имя таблицы>)` — при попытке удалить несуществующую таблицу вызывает ошибку:

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

  - `dropIfExists(<имя таблицы>)` — при попытке удалить несуществующую таблицу ничего не делает, и ошибка не возникает.

4.1.5. Проверка существования таблиц и полей

- Проверка существования таблицы — выполняется вызовом метода `hasTable(<имя таблицы>)` у фасада `Schema`. Метод возвращает `true`, если таблица с указанным *именем* присутствует в базе данных, и `false` — в противном случае. Пример:

  ```
  if (!Schema::hasTable('offers'))
      Schema::create('offers', function (Blueprint $table) { ... });
  ```

- Проверка существования поля — выполняется вызовом метода `hasColumn()` у фасада `Schema`:

  `hasColumn(<имя таблицы>, <имя поля>)`

  Метод возвращает `true`, если поле с указанным *именем* присутствует в таблице с заданным *именем*, и `false` — в противном случае;

- Проверка существования нескольких полей — выполняется вызовом метода `hasColumns()` у фасада `Schema`:

  `hasColumns(<имя таблицы>, <массив с именами полей>)`

  Метод возвращает `true`, если все поля, указанные в *массиве*, присутствуют в таблице с заданным *именем*, и `false` — в противном случае (даже если в таблице нет хотя бы одного поля из *массива*).

4.1.6. Указание базы данных, с которой будут работать миграции

Все операции по созданию, правке и удалению таблиц, рассмотренные в *разд. 4.1.3* и *4.1.4*, выполняются в базе данных, указанной в настройках как используемой по умолчанию (см. *разд. 3.4.2.4*).

Чтобы выполнить эти операции в другой базе, следует указать ее имя в том виде, в котором оно записано в рабочей настройке `connections`:

- либо в защищенном (`protected`) свойстве `connection` класса миграции:

```
return new class extends Migration {
    protected $connection = 'pgsql';
    . . .
};
```

- либо в вызове метода `connection(<имя базы данных>)`. Этот метод вызывается непосредственно у фасада `Schema`, а вызовы методов, создающих, правящих и удаляющих таблицу, записываются вслед за ним. Пример:

```
Schema::connection('pgsql')
      ->create('rubrics', function (Blueprint $table) { ... });
```

Еще можно указать нужную базу данных в командах обработки миграций, о чем будет рассказано далее.

4.1.7. Обработка миграций

4.1.7.1. Применение миграций

Для применения миграций служит команда:

```
php artisan migrate [--database=<имя базы данных>]
[--path=<путь> [--realpath]] ↳
[--seed [--seeder=<имя класса корневого сидера>]] ↳
[--step] [--force] [--pretend]
```

Будут применены все миграции, которые к настоящему времени еще не были применены. Каждая из примененных миграций будет зарегистрирована в журнале миграций отдельной записью.

По умолчанию станут обрабатываться миграции, хранящиеся по пути database\migrations, и все применяемые миграции будут выполнены в рамках единой транзакции.

Поддерживаются следующие командные ключи:

- `--database` — указывает *имя базы данных*, с которой будут работать миграции (если не задан, миграции работают с базой данных по умолчанию или указанной в классе миграции, подробности — в *разд. 4.1.6*);
- `--path` — ищет модули применяемых миграций по указанному *пути* (вместо используемого по умолчанию database\migrations). Можно указать как относительный *путь* (от папки проекта), так и абсолютный, добавив ключ `--realpath`;

- `--seed` — после выполнения миграций выполняет корневой сидер (подробнее о сидерах разговор пойдет позже);
- `--seeder` — задает имя класса выполняемого корневого сидера (если не указан, будет выполнен сидер `DatabaseSeeder`);
- `--step` — выполняет каждую применяемую миграцию в отдельной транзакции (а не все в одной, как обычно), чтобы впоследствии ее можно было индивидуально откатить;
- `--force` — выполняет миграции немедленно, если сайт работает в эксплуатационном режиме (рабочей настройке `env` из модуля config\app.php дано значение `'production'`). Если не указан, и сайт работает в эксплуатационном режиме, перед применением миграций утилита artisan запросит разрешение;
- `--pretend` — выводит на экран все SQL-запросы, отправляемые базе данных при выполнении миграций.

4.1.7.2. Откат миграций, обновление, сброс и очистка базы данных

- Откат миграций — выполняется командой:

```
php artisan migrate:rollback ↳
[--step=<количество откатываемых миграций>] ↳
[--database=<имя базы данных>] [--path=<путь> [--realpath]] ↳
[--force] [--pretend]
```

  При откате очередной миграции описывающая ее запись журнала миграций удаляется.

  По умолчанию обрабатываются миграции, хранящиеся по пути database\migrations, и откатываются все миграции, которые были выполнены в последней транзакции (например, если ранее были применены 3 миграции, а ключ `--step` не указывался, то под откат попадут все 3).

  Поддерживаются следующие ключи:

  - `--step` — указывает количество миграций, которые следует откатить. Позволяет откатить не все миграции, выполненные в последней транзакции, а лишь часть из них (например, указав ключ `--step=1`, можно откатить лишь самую последнюю миграцию);
  - `--database`, `--path`, `--realpath`, `--force` и `--pretend` — описаны в *разд. 4.1.7.1*;

- *сброс базы данных* (откат всех миграций) — выполняется командой:

```
php artisan migrate:reset [--database=<имя базы данных>] ↳
[--path=<путь> [--realpath]] [--force] [--pretend]
```

  Поддерживаемые командные ключи были описаны в *разд. 4.1.7.1*;

- *обновление базы данных* — при котором миграции сначала откатываются, а потом применяются. В ряде случаев позволяет восстановить базу данных в изначальном состоянии. Выполняется командой:

```
php artisan migrate:refresh ↳
[--step=<количество обрабатываемых миграций>] ↳
```

```
[--database=<имя базы данных>] [--path=<путь> [--realpath]] ↳
[--seed [--seeder=<имя класса корневого сидера>]] [--force]
```

По умолчанию обрабатываются *все* миграции без исключения. Также можно указать количество обрабатываемых миграций в ключе `--step`.

Остальные поддерживаемые командные ключи были описаны в *разд. 4.1.7.1*;

- *восстановление базы данных* — при котором из базы удаляются все таблицы (непосредственно, а не путем отката миграций), а потом применяются все миграции. Может пригодиться при восстановлении базы данных в исходное состояние, если операция обновления (см. ранее) не удалась вследствие сбоя при откате какой-либо из миграций или нарушения ссылочной целостности (например, была выполнена попытка удалить первичную таблицу перед вторичной). Выполняется командой:

```
php artisan migrate:fresh [--database=<имя базы данных>] ↳
[--path=<путь> [--realpath]] ↳
[--seed [--seeder=<имя класса корневого сидера>]] [--step] ↳
[--drop-views] [--drop-types] [--force]
```

Поддерживаются ключи:

- `--drop-views` — также удаляет все представления (views);
- `--drop-types` (только PostgreSQL) — также удаляет все типы данных;
- `--database`, `--path`, `--realpath`, `--seed` и `--force` — описаны в *разд. 4.1.7.1*;
- `--seeder` — см. описание команды обновления базы данных.

Может оказаться полезным выполнить *очистку базы данных*, при которой база данных полностью очищается от таблиц. Для этого следует набрать команду:

```
php artisan db:wipe [--database=<имя базы данных>] [--drop-views] ↳
[--drop-types] [--force]
```

Ключи `--database` и `--force` были описаны в *разд. 4.1.7.1*, а ключи `--drop-views` и `--drop-types` — в описании команды восстановления базы данных.

4.1.7.3. Создание журнала миграций и просмотр их состояния

- Создание журнала миграций — таблицы в базе данных, где хранятся информационные записи об их применении, — может понадобиться при случайном удалении этой таблицы. Выполняется командой:

```
php artisan migrate:install [--database=<имя базы данных>]
```

- вывод состояния миграций — их перечня с указанием, были ли они выполнены и, если были, порядкового номера включающей их транзакции. Выполняется командой:

```
php artisan migrate:status [--database=<имя базы данных>] ↳
[--path=<путь> [--realpath]]
```

Ключи `--database`, `--path` и `--realpath` были описаны в *разд. 4.1.7.1*.

4.1.8. Дамп базы данных как альтернатива миграциям

В процессе разработки сайта могут быть созданы десятки, а то и сотни миграций. Если их использовать для создания базы данных на платформе, где будет публиковаться сайт, они будут выполняться очень долго. Для ускорения этого процесса можно использовать *дамп* — текстовый файл, содержащий SQL-команды, которые создают все необходимые структуры: таблицы и индексы.

Дамп генерируется командой:

```
php artisan schema:dump [--database=<имя базы данных>] [--path=<путь>] [--prune]
```

Создаваемый дамп содержит наборы SQL-команд, формирующих:

- все таблицы, поля и индексы, имеющиеся в базе данных на текущий момент;
- журнал миграций со всеми миграциями, присутствующими в журнале оригинальной базы.

Имя файла дампа имеет формат <*имя базы данных*>-schema.dump, где *имя базы данных* берется из рабочей настройки `database.connections`.

Поддерживаются следующие командные ключи:

- `--database` — имя базы данных, дамп которой нужно создать (если не указан, будет создан дамп базы данных, заданной как используемая по умолчанию);
- `--path` — путь к папке, в которой следует сохранить файл с дампом, вместо используемой по умолчанию database\schema (изначально не существует в проекте и создается при первом создании дампа);
- `--prune` — после формирования дампа удалить все миграции.

Создание базы данных на основе дампа осуществляется той же командой `migrate` утилиты artisan (см. *разд. 4.1.7.1*). При этом сначала выполняются SQL-команды из дампа, а потом — не примененные ранее миграции.

Также можно выполнить восстановление базы данных, отдав команду `migrate:fresh` утилиты artisan (см. *разд. 4.1.7.2*).

В обоих командах можно указать командный ключ `--schema-path=`<*путь*>, задающий *путь*, по которому хранится файл с дампом.

4.2. Сидеры

Сидер — это программный модуль, заносящий в таблицы баз данных записи со строго определенным содержимым. Сидеры применяются для заполнения базы отладочными записями.

Сидеры реализуются в виде подклассов класса `Illuminate\Database\Seeder`, объявляются в пространстве имен `Database\Seeders`, соответственно их модули хранятся в папке database\seeders.

Корневой сидер, непосредственно запускаемый утилитой artisan при отдаче соответствующих команд, генерируется при создании нового проекта и носит имя `DatabaseSeeder`. Дополнительно можно создать произвольное количество *подчиненных сидеров*, запускаемых из корневого.

4.2.1. Использование корневого сидера

Корневой сидер `DatabaseSeeder` содержит изначально «пустой» метод `run()`, в котором и записывается код, создающий записи.

В листинге 4.1 приведен код сидера, создающего пользователя `admin`.

Листинг 4.1. Сидер, создающий нового пользователя

```
namespace Database\Seeders;

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

use Illuminate\Support\Facades\Hash;
use App\Models\User;

class DatabaseSeeder extends Seeder {
    public function run() {
        User::create(['name' => 'admin', 'email' => 'admin@bboard.ru',
                      'password' => Hash::make('admin')]);
    }
}
```

Можно создать произвольное количество записей со случайным содержимым. Пример создания трех пользователей со случайными именами, адресами электронной почты и одинаковым паролем `user`:

```
use Illuminate\Support\Str;
. . .
$password = Hash::make('user');
for ($i = 0; $i < 3; $i++)
    User::create(['name' => Str::random(10),
                  'email' => Str::random(10) . '@bboard.ru',
                  'password' => $password]);
```

Метод `random()`, вызываемый у фасада `Str`, выдает случайную строку заданной в параметре длины (более подробное описание — в *главе 14*).

В процессе работы сидера генерируются события модели (описаны в *главе 22*). Чтобы подавить их генерирование (например, для повышения производительности), следует присоединить к классу сидера трейт `Illuminate\Database\Console\Seeds\WithoutModelEvents` (в коде сидера уже присутствует выражение, выполняющее его импорт). Пример:

```
class DatabaseSeeder extends Seeder {
    use WithoutModelEvents;
    . . .
}
```

4.2.2. Использование подчиненных сидеров

Если база данных содержит множество таблиц, писать код, заносящий в них записи, в одном корневом сидере неудобно. Лучше разнести код, добавляющий записи в от-

дельные таблицы, по отдельным подчиненным сидерам, а в корневой сидер вставить код, вызывающий их. Выполняется это в три этапа.

1. Создание подчиненного сидера — отдачей команды формата:

   ```
   php artisan make:seeder <имя класса сидера>
   ```

   По принятым соглашениям, имя класса сидера должно заканчиваться словом `Seeder` (например, для заполнения таблицы пользователей следует создать сидер `UserSeeder`).

2. Написание подчиненного сидера — по тем же принципам, по которым пишется корневой сидер. Пример:

   ```
   use Illuminate\Support\Facades\DB;
   use Illuminate\Support\Facades\Hash;
   class UserSeeder extends Seeder {
       public function run() {
           DB::table('users')
             ->insert(['name' => 'editor',
                       'email' => 'editor@bboard.ru',
                       'password' => Hash::make('editor')]);
       }
   }
   ```

   Фасад `DB` открывает прямой, без посредства моделей, доступ к подсистеме, работающей с базами данных. Метод `table()` задает таблицу базы данных, а метод `insert()` добавляет в нее новую запись и записывает в ее поля значения из переданного массива.

3. Указание корневому сидеру вызвать подчиненный сидер — добавлением в метод `run()` класса корневого сидера вызова метода `call()`, унаследованного от суперкласса:

   ```
   call(<массив с именами классов запускаемых подчиненных сидеров>)
   ```

   Пример:

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

В качестве корневого можно использовать любой из подчиненных сидеров, указав его в ключе `--seeder` или `--class` утилиты artisan. Это может понадобиться в том случае, если требуется выполнить только один сидер.

4.2.3. Выполнение сидеров

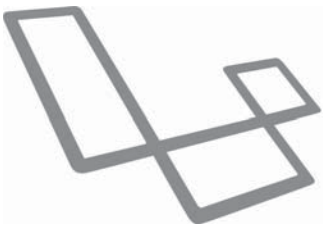
Выполнение сидеров запускается набором команды:

```
php artisan db:seed [--database=<имя базы данных>] ↳
[--class=<имя класса корневого сидера>] [--force]
```

Ключи `--database` и `--force` были описаны в *разд. 4.1.7.1*, а ключ `--class` аналогичен по назначению ключу `--seeder` (см. *разд. 4.1.7.2*).

Выполнить сидеры можно также попутно, при выполнении команд утилиты artisan, описанных в *разд. 4.1.7*.

ГЛАВА 5



Модели: базовые инструменты

Модель — программный модуль, служащий для взаимодействия с определенной таблицей базы данных (носящей название *обслуживаемой*): извлечения значений полей, добавления, правки и удаления записей. Также модель предоставляет прямой доступ к *построителю запросов*, посредством которого производится выборка записей.

Отдельный объект модели хранит значения полей отдельной записи обслуживаемой таблицы, позволяет обратиться к значениям полей через одноименные свойства и предоставляет ряд методов для обработки записи: сохранения, удаления и др.

5.1. Создание моделей

Модель создается командой формата:

```
php artisan make:model <имя класса модели> [--migration] [--seed] ↵
[--controller [--resource [--api]] [--requests]] [--policy] [--all] ↵
[--pivot] [--force]
```

Поддерживаются следующие ключи:

- `--migration` — дополнительно создает миграцию, формирующую обслуживаемую моделью таблицу. Создаваемая таблица получит имя, совпадающее с *именем класса модели* во множественном числе, а сама миграция — имя формата:

  *<год>*_*<№ месяца>*_*<число>*_*<часы>*<*минуты*>*<секунды>*_create_↵

  *<имя таблицы>*_table.php

  В метод `up()` класса миграции будет добавлен код, создающий эту таблицу и добавляющий в нее ключевое поле, поля отметок создания и правки, а в метод `down()` — код, удаляющий эту таблицу (как при отдаче команды `make:migration` с ключом `--create`, подробнее — в *разд. 4.1.1*):

- `--seed` — дополнительно создает сидер с именем формата *<имя класса модели>*`Seeder`;

- `--controller` — дополнительно создает контроллер с именем формата *<имя класса модели>*`Controller`. Дополнительно можно указать следующие ключи:

- `--resource` — создает ресурсный контроллер (подробнее — *в главе 9*);
- `--api` — при использовании с ключом `--resource` — создает ресурсный API-контроллер;
- `--requests` — при использовании с ключом `--resource` — дополнительно создает формальные запросы (подробнее — в *главе 10*) для использования в операциях сохранения новой и исправления существующей записи. Классы создаваемых формальных запросов будут иметь имена формата соответственно `Store<`*имя класса модели*`>Request` и `Update<`*имя класса модели*`>Request`;

- ❒ `--policy` — дополнительно создает политику для модели, давая ей имя формата `<`*имя класса модели*`>Policy`;
- ❒ `--all` — дополнительно создает сразу миграцию, сидер, ресурсный контроллер и политику;
- ❒ `--pivot` — создает расширенную связующую модель (о связующих моделях и связях «многие-со-многими» будет рассказано далее). Если также было указано создание миграции, формируемая ею таблица получит имя, совпадающее с *именем модели*;
- ❒ `--force` — принудительно создает модель, даже если одноименный модуль уже существует.

5.2. Класс модели и соглашения по умолчанию

Класс модели объявляется в пространстве имен `App\Models` (в версиях Laravel, предшествовавших 7, модели объявлялись в пространстве имен `App`) и является производным от класса `Illuminate\Database\Eloquent\Model`. Суперкласс предоставляет все необходимые инструменты как для работы с записью, хранящейся в объекте модели, так и для взаимодействия с построителем запросов.

Изначальный класс модели «пуст» и содержит лишь трейт `Illuminate\Database\Eloquent\Factories\HasFactory`. Последний используется при автоматизированном тестировании (которое в этой книге не описывается), так что его можно удалить. Пример «пустого» класса модели показан в листинге 1.4.

Модель работает в соответствии со следующими соглашениями по умолчанию:

- ❒ база данных — задействуется указанная в настройках проекта как используемая по умолчанию (см. *разд. 3.4.2.4*);
- ❒ обслуживаемая моделью таблица — должна иметь имя, совпадающее с именем класса модели во множественном числе (например, модель `Rubric` будет обслуживать таблицу `rubrics`).

  Если имя класса модели состоит из нескольких слов, набранных вплотную, без пробелов между ними, и начинающихся с прописных букв (стиль именования *PascalCase*), имя таблицы должно представлять собой то же имя, записанное согласно стилю snake_case (так, модель `BbOffer` будет обслуживать таблицу `bb_offers`);
- ❒ имя ключевого поля — `id`;
- ❒ тип ключевого поля — целочисленный автоинкремент;
- ❒ поля отметок создания и правки — должны присутствовать;

- имя поля отметки создания — `created_at`;
- имя поля отметки правки — `updated_at`;
- формат записи временны́х отметок в эти поля — используемый по умолчанию в соответствующей СУБД.

Если обслуживаемая таблица не удовлетворяет этим соглашениям, ее параметры можно указать в свойствах класса модели, описываемых далее.

> ***Полезно знать***
> Подсистема моделей, встроенная в Laravel, носит название Eloquent.

5.3. Параметры модели

5.3.1. Параметры полей модели

Параметры полей таблицы заносятся в защищенные свойства класса модели:

- `fillable` — массив с именами полей, доступных для массового присваивания:

```
class Bb extends Model {
    protected $fillable = ['title', 'content', 'price'];
}
```

  Значение по умолчанию: «пустой» массив;

- `guarded` — массив с именами полей, наоборот, *не* доступных для массового присваивания:

```
class Bb extends Model {
    protected $guarded = ['id', 'created_at', 'updated_at'];
}
```

  Значение по умолчанию: массив с единственным элементом `"*"` (все поля недоступны для массового присваивания).

> ***Следует указать только одно из свойств: или `fillable`, или `guarded`***
> Либо занести в свойство `fillable` массив доступных полей (тогда не указанные в массиве поля не будут доступными), либо в свойство `guarded` — массив недоступных полей (тогда все поля не из этого массива станут доступными).

- `attributes` — ассоциативный массив со значениями по умолчанию, заносимыми в поля сразу при создании записи. Ключи элементов массива должны соответствовать полям, а значения элементам укажут значения этих полей по умолчанию. Пример:

```
class Bb extends Model {
    protected $attributes = ['price' => 100.0, 'publish' => true];
}
```

5.3.2. Параметры обслуживаемой таблицы

Если обслуживаемая таблица не соответствует соглашениям, приведенным в *разд. 5.2*, следует явно указать ее параметры в следующих свойствах класса модели:

- `connection` — защищенное, имя базы данных из числа приведенных в настройках проекта (см. *разд. 3.4.2.4*);
- `table` — защищенное, имя обслуживаемой таблицы;
- `primaryKey` — защищенное, имя ключевого поля;
- `keyType` — защищенное, наименование типа ключевого поля;
- `incrementing` — общедоступное (`public`), если `true`, ключевое поле является автоинкрементным, и беспокоиться о занесении в него уникальных значений не нужно (поведение по умолчанию). Если `false`, ключевое поле не является автоинкрементным, и уникальные значения в него нужно заносить самостоятельно;
- `timestamps` — общедоступное, если `true`, в обслуживаемой таблице есть поля отметок создания и правки, и Laravel должен заносить в них значения (поведение по умолчанию). Если `false`, таких полей в таблице нет;
- `dateFormat` — защищенное, строка с форматом значений, заносимых в поля временны́х отметок.

Остальные параметры задаются в общедоступных константах класса модели:

- `CREATED_AT` — имя поля отметки создания;
- `UPDATED_AT` — имя поля отметки правки.

Пример:

```
class Rubric extends Model {
    protected $connection = 'pgsql';
    protected $table = 'rubric_list';
    protected $primaryKey = 'rubric_abbr';
    protected $keyType = 'string';
    public $incrementing = false;
    protected $dateFormat = 'U';
    const CREATED_AT = 'added';
    const UPDATED_AT = 'edited';
}
```

5.3.3. Параметры преобразования типов

Перед выдачей значения, считанного из базы данных, Laravel преобразует его в наиболее подходящий тип из числа поддерживаемых PHP. Однако иногда возникает необходимость преобразовать считанное значение в какой-либо другой тип (например, значение вещественного поля — в целочисленный тип).

Параметры преобразования типов задаются в защищенном свойстве `casts`. Его значением должен быть ассоциативный массив, ключи элементов которого должны соответствовать именам полей, а значения элементов укажут наименования типов, в которые следует преобразовать значения этих полей, из числа следующих:

- `'string'` — обычная строка;
- полное имя класса `Illuminate\Database\Eloquent\Casts\AsStringable` — строка, являющаяся объектом класса `Stringable` (описан в *главе 14*):

```
use Illuminate\Database\Eloquent\Casts\AsStringable;
class Bb extends Model {
    protected $casts = ['content' => AsStringable::class];
    . . .
}
. . .
$bb = Bb::find(2);
$content = $bb->content->trim()->ucfirst();
```

- `'int'` или `'integer'` — целое число:

```
class Bb extends Model {
    protected $casts = ['price' => 'integer'];
    . . .
}
```

- `'float'`, `'double'` или `'real'` — вещественное число;
- `'decimal:<`*количество цифр после запятой*`>'` — вещественное число высокой точности;
- `'datetime[:<`*формат*`>]'` или `'custom_datetime[:<`*формат*`>]'` — временна́я отметка в виде объекта класса `Carbon`. Можно указать *формат*, в котором извлеченное из базы значение временно́й отметки будет сериализоваться в JSON (на формат, в котором значение хранится в поле базы данных, это не повлияет). Пример:

```
class Bb extends Model {
    protected $casts = ['created_at' => 'datetime:Y-m-d'];
    . . .
}
```

> ***Полное описание класса Carbon...***
>
> ...можно найти на «домашнем» сайте **https://carbon.nesbot.com/**. Этот класс является производным от стандартного класса `DateTime` PHP.

- `'timestamp'` — временна́я отметка в формате UNIX (количество секунд, прошедших с полуночи 1 января 1970 года);
- `'date[:<`*формат*`>]'` — то же самое, что и `datetime`, только с временем 00:00:00 (фактически дата без времени);
- `'immutable_datetime'` или `'immutable_custom_datetime'` — то же самое, что `datetime`, только неизменяемая;
- `'immutable_date'` — то же самое, что `date`, только неизменяемая;
- `'bool'` или `'boolean'` — логический:

```
class User extends Model {
    protected $casts = ['is_admin' => 'boolean'];
    . . .
}
```

- полное имя типизированного перечисления[1] — соответствующий вариант этого перечисления:

[1] Поддержка перечислений появилась в PHP 8.1.

```
enum BbType: int {
    case BUY = 1;
    case SELL = 2;
}
. . .
class Bb extends Model {
    protected $casts = ['type' => BbType::class];
    . . .
}
. . .
$bb = Bb::find(10);
if ($bb->type == BbType::SELL) {
    $type_desc = 'Объявление о продаже'
} else {
    $type_desc = 'Объявление о покупке';
}
```

- `'array'` или `'json'` — массив. В такой тип могут быть преобразованы данные, хранящиеся в текстовых и JSON-полях. Пример:

```
class Bb extends Model {
    protected $casts = ['address' => 'array'];
    . . .
}
. . .
$bb = Bb::find(3);
$city = $bb->address['city'];
```

Недостатком полученного массива является невозможность изменения значений элементов такого массива. Например, следующий код вызовет ошибку:

```
$bb->address['city'] = 'Волжский';
```

Чтобы изменить значения элементов такого массива, следует создать его копию, изменить значения *в ней*, после чего занести измененную копию массива в поле модели:

```
$addr = $bb->address;
$addr['city'] = 'Волжский';
$bb->address = addr;
```

- полное имя класса `Illuminate\Database\Eloquent\Casts\AsArrayObject` — объект класса `ArrayObject`, встроенного в PHP, который имеет функциональность массива, но лишен описанного ранее недостатка:

```
use Illuminate\Database\Eloquent\Casts\AsArrayObject;
class Bb extends Model {
    protected $casts = ['address' => AsArrayObject::class];
    . . .
}
. . .
$bb = Bb::find(3);
$bb->address['city'] = 'Волжский';                    // Успешно
```

- `'object'` — объект класса `stdClass`;
- `'collection'` или полное имя класса `Illuminate\Database\Eloquent\Casts\AsCollection` — коллекция в виде объекта класса `Collection` (описан в *главе 15*)
- `'encrypted'` — элементарное значение, зашифрованное средствами Laravel (описаны в *разд. 26.4*);
- `'encrypted:array'` — зашифрованный ассоциативный массив PHP;
- полное имя класса `Illuminate\Database\Eloquent\Casts\AsEncryptedArrayObject` — зашифрованный объект класса `ArrayObject`;
- `'encrypted:object'` — зашифрованный объект класса `stdClass`;
- `'encrypted:collection'` или полное имя класса `Illuminate\Database\Eloquent\Casts\AsEncryptedCollection` — зашифрованная коллекция `Collection`.

5.3.4. Реализация «мягкого» удаления записей в моделях

Чтобы иметь возможность «мягко» удалять записи модели, помимо создания в таблице поля отметки удаления (см. *разд. 4.1.3.2*) в сам класс модели следует добавить трейт `Illuminate\Database\Eloquent\SoftDeletes`. Пример:

```
use Illuminate\Database\Eloquent\SoftDeletes;
class Bb extends Model {
    use SoftDeletes;
    . . .
}
```

5.4. Создание связей между моделями

5.4.1. Связь «один-со-многими»

Для создания между моделями связи «один-со-многими» в их классах объявляются два общедоступных и не принимающих параметров метода:

1. В классе первичной модели — метод, создающий связь со вторичной моделью (назовем эту связь «прямой»). Обычно имя этого метода совпадает с именем связанной вторичной таблицы, хотя может быть произвольным. Метод должен возвращать объект «прямой» связи, возвращенный вызовом метода `hasMany()` модели:

   ```
   hasMany(<имя класса связываемой вторичной модели>[,
           <имя поля внешнего ключа вторичной модели>=null[,
           <имя ключевого поля первичной модели>=null]])
   ```

   Если *имя поля внешнего ключа* не указано, Laravel предполагает, что вторичная модель содержит поле внешнего ключа с именем формата *<имя первичной модели>*`_id` (например, для связи с первичной моделью `Rubric` вторичная модель использует поле `rubric_id`). Если не указано *имя ключевого поля*, будет использовано ключевое поле `id` или заданное в свойстве `primaryKey` первичной модели.

2. В классе вторичной модели — метод, создающий связь с первичной моделью («обратную»). Обычно его имя совпадает с именем связанной первичной модели (хотя, опять же, может быть любым). Метод должен возвращать объект «обратной» связи, возвращенный методом `belongsTo()` модели:

```
belongsTo(<имя класса связываемой первичной модели>[,
          <имя поля внешнего ключа вторичной модели>=null[,
          <имя ключевого поля первичной модели>=null]])
```

   Имена поля внешнего ключа вторичной модели и *ключевого поля первичной модели* по умолчанию вычисляются так же, как и в случае метода `hasMany()`.

Пример создания связи «один-со-многими» между первичной моделью `Rubric` (перечень рубрик) и вторичной `Bb` (перечень объявлений) при условии, что имена полей внешнего ключа и ключевого соответствуют принятым соглашениям:

```
// Миграция, создающая таблицу rubrics
Schema::create('rubrics', function (Blueprint $table) {
    $table->id();
    . . .
});

// Миграция, создающая таблицу bbs
Schema::create('bbs', function (Blueprint $table) {
    . . .
    $table->foreignId('rubric_id')->constrained()->cascadeOnDelete();
    . . .
});

// Модель Rubric
use App\Models\Bb;
class Rubric extends Model {
    public function bbs() {
        return $this->hasMany(Bb::class);
    }
}

// Модель Bb
use App\Models\Rubric;
class Bb extends Model {
    public function rubric() {
        return $this->belongsTo(Rubric::class);
    }
}
```

Пример создания аналогичной связи в случае, если имена полей внешнего ключа и ключевого *не* соответствуют принятым соглашениям:

```
Schema::create('rubrics', function (Blueprint $table) {
    $table->string('rubric_key', 5)->primary();
    . . .
});
```

```
Schema::create('bbs', function (Blueprint $table) {
    . . .
    $table->string('rubric', 5);
    $table->foreign('rubric')->references('rubric_key')->on('rubrics')
                             ->cascadeOnDelete();
    . . .
});

use App\Models\Bb;
class Rubric extends Model {
    public function bbs() {
        return $this->hasMany(Bb::class, 'rubric', 'rubric_key');
    }
}

use App\Models\Rubric;
class Bb extends Model {
    protected $primaryKey = 'rubric_key';
    protected $keyType = 'string';
    protected $incrementing = false;

    public function rubric() {
        return $this->belongsTo(Rubric::class, 'rubric', 'rubric_key');
    }
}
```

5.4.2. Связь «один-с-одним из многих»

Связь «один-с-одним из многих» при обращении со стороны первичной модели из всех имеющихся связанных записей вторичной модели выдает лишь одну, соответствующую указанному критерию. В качестве примера можно привести связь, которая при обращении из какой-либо рубрики выдает наиболее «свежее» объявление, находящееся в этой рубрике, или объявление с наивысшей заявленной ценой.

Связь такого рода создается почти так же, как и «прямая» связь типа «один-со-многими» (см. *разд. 5.4.1*), но за двумя отличиями:

- в методе, формирующем связь со вторичной моделью, — сама связь создается вызовом метода `hasOne()` модели, чей формат вызова аналогичен таковому у метода `hasMany()` (см. *разд. 5.4.1*);
- у объекта «прямой» связи, возвращенного методом `hasOne()`, — в зависимости от того, какую запись вторичной модели предполагается получать посредством создаваемой связи, следует вызвать один из следующих методов:
  - `latestOfMany([<`*имя поля*`>='id'])` — предполагается получить запись, в которой поле с указанным *именем* хранит максимальное значение.

    По умолчанию этот метод просматривает значения ключей записей, хранящихся в поле `id`. Поскольку при добавлении записей в это поле заносятся последовательно увеличивающиеся целочисленные номера, запись, имеющая максимальное значение ключа, является наиболее «свежей»;

- `oldestOfMany([<имя поля>='id'])` — предполагается получить запись, в которой поле с указанным *именем* хранит минимальное значение;
- `ofMany()` — позволяет указать более сложное условие для выборки получаемой записи. Поддерживает два формата вызова:

```
ofMany([<имя поля>='id'[, <агрегатная функция>='MAX']])
ofMany(<массив полей и агрегатных функций>,
       <анонимная функция, задающая условия фильтрации>)
```

  Первый формат позволяет указать *имя поля*, значение которого будет проверяться, и *агрегатную функцию*, задающую род проверки: `'MAX'` — будет извлекаться запись с максимальным значением в указанном поле, `'MIN'` — с минимальным значением.

  Второй формат позволяет выполнить проверку, во-первых, по произвольному количеству полей (например, выбрать наиболее «свежее» объявление с минимальной ценой) и, во-вторых, с учетом указанных условий фильтрации записей (скажем, отбирать только записи, созданные в течение текущего месяца).

  Ключи элементов в указанном ассоциативном *массиве* должны соответствовать проверяемым полям модели, а значения этих элементов должны представлять собой агрегатные функции. Сначала проверяются значения первого из указанных в *массиве* полей, если будет найдено несколько записей с одинаковым значением, проверяются значения второго поля и т. д.

  Анонимная функция должна принимать в качестве единственного параметра объект построителя запросов и задавать у него необходимые условия фильтрации, вызывая методы, описанные в *разд. 7.3.4*.

Пример создания между первичной моделью `Rubric` и вторичной моделью `Bb` связи «один-с-одним из многих», выдающей последнее добавленное объявление:

```
class Rubric extends Model {
    . . .
    public function latestBb() {
        return $this->hasOne(Bb::class)->latestOfMany();
    }
}
```

Аналогичную связь можно создать следующим образом:

```
return $this->hasOne(Bb::class)->ofMany('created_at', 'MAX');
```

Пример создания связи, которая выдает последнее добавленное объявление с минимальной ценой, созданное в течение текущего месяца (функция-хелпер `now()` выдает текущую дату в виде объекта класса `Carbon`, а свойство `month` этого класса хранит текущий месяц):

```
class Rubric extends Model {
    . . .
    public function latestMinPriceBbOnLastMonth() {
        return $this->hasOne(Bb::class)->ofMany(
            ['price' => 'MIN', 'created_at' => 'MAX'],
```

```
            function ($query) {
                $query->whereMonth('created_at', now()->month);
            }
        );
    }
}
```

5.4.3. Связь «один-с-одним»

Для реализации между моделями связи «один-с-одним» нужно объявить в них два метода, также общедоступных и не принимающих параметров:

1. В классе первичной модели — метод, формирующий «прямую» связь со вторичной моделью. Обычно его имя совпадает с именем связанной вторичной модели. Метод должен возвращать объект «прямой» связи, возвращенный вызовом метода `hasOne()` модели, описанного в *разд. 5.4.2*.
2. В классе вторичной модели — такой же метод, формирующий «обратную» связь, что и во вторичной модели, связанной связью «один-со-многими» (см. *разд. 5.4.1*).

Пример создания связи «один-с-одним» между стандартной первичной моделью `User` (список зарегистрированных пользователей) и вторичной моделью `Account` (дополнительные сведения о пользователе):

```
// Миграция, создающая таблицу accounts
Schema::create('accounts', function (Blueprint $table) {
    . . .
    $table->foreignId('user_id')->constrained()->cascadeOnDelete();
    . . .
});

// Модель User
use App\Models\Account;
class User extends Model {
    public function account() {
        return $this->hasOne(Account::class);
    }
}

// Модель Account
use App\Models\User;
class Account extends Model {
    public function user() {
        return $this->belongsTo(User::class);
    }
}
```

5.4.4. Связь «многие-со-многими»

Связь «многие-со-многими» реализуется сложнее, чем ранее рассмотренные виды связей. Для этого необходимо:

1. Создать *связующую таблицу* — содержащую два поля внешнего ключа:
   - первое — для хранения ключа связанной записи первой из связываемых таблиц;
   - второе — для хранения ключа связанной записи второй из связываемых таблиц.

   По соглашению имена этих полей должны соответствовать формату *<имя связываемой модели>*_id. Сама связующая таблица, также по соглашению, должна иметь имя, составленное из имен связываемых моделей, выстроенных по алфавиту и разделенных символом подчеркивания. Например, для связывания моделей Spare и Machine следует создать связующую таблицу machine_spare с полями machine_id и spare_id.

   Связующая таблица не обязана содержать полей отметок создания и правки записи, равно как и ключевого поля — обычно необходимости в них нет (за исключением каких-либо специфических случаев).

   Связующая таблица создается так же, как и любые другие таблицы, — с помощью миграции. Создавать модель для связующей таблицы необязательно.

2. В первой связываемой модели — объявить общедоступный, не принимающий параметров метод, формирующий связь между таблицей, обслуживаемой текущей моделью, и связующей таблицей. Как правило, его имя совпадает с именем второй связываемой таблицы. Метод должен возвращать объект связи, возвращенный вызовом метода belongsToMany() модели:

   ```
   belongsToMany(<имя класса второй связываемой модели>[,
                 <имя связующей таблицы>=null[,
                 <имя поля связующей таблицы, хранящей номер связанной ↩
                 записи текущей таблицы>=null[,
                 <имя поля связующей таблицы, хранящей номер связанной ↩
                 записи второй таблицы>=null[,
                 <имя ключевого поля текущей таблицы>=null[,
                 <имя ключевого поля второй таблицы>=null]]]]])
   ```

   Если *имена полей связующей таблицы* не указаны, Laravel предполагает, что эти поля имеют имена формата *<имя связываемой модели>*_id. Если не заданы *имена ключевых полей*, будут использованы поля id или заданные в свойствах primaryKey моделей.

3. Во второй связываемой модели — объявить аналогичный метод.

Пример установления связи «многие-со-многими» между моделями Machine (машина) и Spare (отдельная деталь), если имена связующей таблицы и полей соответствуют соглашениям:

```
// Миграция, создающая таблицу machines
Schema::create('machines', function (Blueprint $table) {
    $table->id();
    $table->string('name', 30);
    $table->timestamps();
});

// Миграция, создающая таблицу spares
Schema::create('spares', function (Blueprint $table) {
    $table->id();
```

```
    $table->string('name', 30);
    $table->timestamps();
});

// Миграция, создающая связующую таблицу machine_spare
Schema::create('machine_spare', function (Blueprint $table) {
    $table->foreignId('machine_id')->constrained()->cascadeOnDelete();
    $table->foreignId('spare_id')->constrained()->cascadeOnDelete();
});

// Модель Machine
use App\Models\Spare;
class Machine extends Model {
    public function spares() {
        return $this->belongsToMany(Spare::class);
    }
}

// Модель Spare
use App\Models\Machine;
class Spare extends Model {
    public function machines() {
        return $this->belongsToMany(Machine::class);
    }
}
```

Установление аналогичной связи в случае, если имена связующей таблицы и полей *не* соответствуют соглашениям:

```
// Миграция, создающая таблицу machines
Schema::create('machines', function (Blueprint $table) {
    $table->id('machine_key');
    . . .
});

// Миграция, создающая таблицу spares
Schema::create('spares', function (Blueprint $table) {
    $table->id('spare_key');
    . . .
});

// Миграция, создающая связующую таблицу ms
Schema::create('ms', function (Blueprint $table) {
    $table->foreignId('machine')->constrained('machines', 'machine_key')
                                ->cascadeOnDelete();
    $table->foreignId('spare')->constrained('spares', 'spare_key')
                               ->cascadeOnDelete();
});

// Модель Machine
use App\Models\Spare;
```

```
class Machine extends Model {
    protected $primaryKey = 'machine_key';
    public function spares() {
        return $this->belongsToMany(Spare::class, 'ms', 'machine',
                                    'spare', 'machine_key', 'spare_key');
    }
}

// Модель Spare
use App\Models\Machine;
class Spare extends Model {
    protected $primaryKey = 'spare_key';
    public function machines() {
        return $this->belongsToMany(Machine::class, 'ms', 'spare',
                                    'machine', 'spare_key',
                                    'machine_key');
    }
}
```

В связующую таблицу можно добавить дополнительные поля. Пример объявления дополнительного поля `cnt`, хранящего количество деталей в машине:

```
Schema::create('ms', function (Blueprint $table) {
    . . .
    $table->unsignedSmallInteger('cnt')->default(1);
});
```

Каждый объект связанной модели будет поддерживать свойство `pivot`, хранящее объект с записью связующей таблицы (подробности — в *главах 6* и *7*). Обратившись к этому объекту, можно извлечь значения полей записи из связующей таблицы.

Однако имейте в виду, что этот объект генерируется самим фреймворком и хранит только поля с ключами связанных записей (в нашем случае — поля `machine_id` и `spare_id`). Если же связующая модель имеет дополнительные поля, они в объект из свойства `pivot` по умолчанию не заносятся.

Можно задать дополнительные параметры связи «многие-со-многими», вызвав у объекта связи, возвращенного методом `belongsToMany()`, следующие методы:

- `withPivot()` — указывает дополнительные поля связующей таблицы, которые должны содержаться в объекте из свойства `pivot` связанной модели. Форматы вызова метода:

  ```
  withPivot(<имя поля 1>, <имя поля 2> ... <имя поля n>)
  withPivot(<массив с именами полей>)
  ```

  Возвращаемый результат — текущий объект связи;

- `withTimestamps()` — указывает обрабатывать поля отметок создания и правки записи связующей таблицы (если таковые поля в ней присутствуют) и добавляет их в объект из свойства `pivot`. Формат вызова:

  ```
  withTimestamps([<имя поля отметки создания>=null[,
                  <имя поля отметки правки>=null]])
  ```

Если *имена полей* не заданы, Laravel предположит, что они имеют имена по умолчанию: `created_at` и `updated_at`. Возвращаемый результат — текущий объект связи;

- `as(<имя свойства>)` — задает другое имя для свойства, хранящего объект с записью связующей таблицы (вместо `pivot`). Возвращает текущий объект связи.

Пример выборки из связующей таблицы поля `cnt`, полей с отметками и указания для свойства, хранящего объект записи связующей таблицы, имени `connector`:

```
return $this->belongsToMany(Spare::class)->as('connector')
            ->withPivot('cnt')->withTimestamps();
```

5.4.4.1. Использование связующих моделей

Если связующая таблица, помимо полей внешних ключей, содержит дополнительные поля, удобно создать для нее отдельную *связующую модель*. Для этого следует указать ключ `--pivot` у команды `make:model` утилиты artisan:

```
php artisan make:model <имя класса связующей модели> --pivot
```

Связующая модель имеет следующие отличия от обычной:

- наследует от суперкласса `Illuminate\Database\Eloquent\Relations\Pivot`;
- помечена как не содержащая автоинкрементного ключевого поля (в суперклассе свойству `incrementing` дано значение `false`).

  Если таблица, обслуживаемая такой моделью, содержит автоинкрементное ключевое поле, в классе модели этому свойству следует явно присвоить `true`;
- помечает все свои поля как доступные для массового присваивания (в суперклассе свойству `guarded` в качестве значения дан «пустой» массив).

Если связующая таблица не содержит полей отметок создания и правки записи, рекомендуется пометить модель как не содержащую этих отметок, дав свойству `timestamps` значение `false`.

Связи с обеими связываемыми моделями в связующей модели устанавливаются посредством метода `belongsTo()`.

Пример связующей модели `MachineSpare` приведен в листинге 5.1.

Листинг 5.1. Пример связующей модели

```
namespace App;
use Illuminate\Database\Eloquent\Relations\Pivot;
use App\Models\Machine;
use App\Models\Spare;

class MachineSpare extends Pivot {
    public $timestamps = false;

    public function machine() {
        return $this->belongsTo(Machine::class);
    }
```

```
    public function spare() {
        return $this->belongsTo(Spare::class);
    }
}
```

При создании связи в связываемых моделях следует явно указать, что для обработки связующей таблицы должна использоваться связующая модель, и задать имена полей, которые должны присутствовать в объекте из свойства `pivot`.

Связующая модель задается в вызове метода `using(<имя класса связующей модели>)` объекта связи. Извлекаемые из связующей модели поля задаются в методе `withPivot()`, описанном в *разд. 5.4.4*. Пример:

```
use App\Models\MachineSpare;
class Machine extends Model {
    public function spares() {
        return $this->belongsToMany(Spare::class)
                    ->using(MachineSpare::class)->withPivot('cnt');
    }
}
```

5.4.5. Пометка записи первичной модели как исправленной при правке или удалении связанных записей вторичной модели

По умолчанию при правке или удалении записей вторичной модели Laravel не помечает связанную с ними запись первичной модели как исправленную, обновляя в ней значение поля отметки правки. Чтобы указать фреймворку помечать в таких случаях запись первичной модели как исправленную, следует:

1. Объявить в классе *вторичной* модели защищенное поле `touches`.
2. Присвоить этому свойству массив с именами «обратных» связей, указывающих на первичные модели, записи которых нужно помечать как исправленные.

Пример:

```
class Bb extends Model {
    protected $touches = ['rubric'];
}
```

5.4.6. Сквозная связь «один-со-многими»

Предположим, есть первичная модель `Rubric`, связанная со вторичной моделью `Bb`, которая, в свою очередь, выступая в качестве первичной, связана со вторичной моделью `Offer` (предложения). Laravel позволяет создать сквозную связь «один-со-многими» между моделями `Rubric` (назовем ее *начальной*) и `Offer` (*конечной*) через модель `Bb` (*промежуточную*).

Для этого в *начальной* модели объявляется метод, создающий такую связь. Обычно его имя совпадает с именем связанной конечной таблицы. Метод должен возвращать объект связи, возвращенный вызовом метода `hasManyThrough()` модели:

```
hasManyThrough(<имя класса конечной модели>,
               <имя класса промежуточной модели>[,
               <имя поля внешнего ключа промежуточной модели>=null[,
               <имя поля внешнего ключа конечной модели>=null[,
               <имя ключевого поля начальной модели>=null[,
               <имя ключевого поля промежуточной модели>=null]]]])
```

Если какое-либо из *имен полей* не указано, фреймворк предполагает, что оно соответствует принятым соглашениям.

Пример:

```
use App\Models\Bb;
use App\Models\Offer;
class Rubric extends Model {
    public function offers() {
        return $this->hasManyThrough(Offer::class, Bb::class);
    }
}
```

5.4.7. Сквозная связь «один-с-одним»

Сквозная связь «один-с-одним» устанавливается так же, как и аналогичная связь «один-со-многими» (см. *разд. 5.4.6*), только для создания связи следует вызвать метод `hasOneThrough()`. Формат его вызова такой же, как и у метода `hasManyThrough()`.

5.4.8. Записи-заглушки

Если запись вторичной модели не связана с какой-либо записью первичной модели, а также если запись первичной модели не связана ни с одной записью вторичной модели в случае связи «один-с-одним», обычной или сквозной, при попытке извлечь связанную запись будет получено значение `null`. Это приводит к необходимости выполнять проверку, действительно ли обрабатываемая запись связана с другой записью, — поскольку попытка обращения к отсутствующей связанной записи приведет к ошибке.

Можно указать фреймворку в таких случаях выдавать не `null`, а *запись-заглушку*, представляющую собой объект связанной модели, который может быть как «пустым», так и содержать какие-либо данные. Для этого у объекта связи, возвращенного методом `belongsTo()`, `hasOne()` или `hasOneThrough()`, следует вызвать метод `withDefault()`, поддерживающий три формата вызова:

```
withDefault()
withDefault(<ассоциативный массив со значениями полей записи>)
withDefault(<анонимная функция>)
```

Первый формат вызова метода создает «пустую» запись — объект связанной модели, поля которого не содержат никаких данных (или значения по умолчанию, если они были указаны, — см. *разд. 5.3.1*):

```
class Bb extends Model {
    public function user() {
```

```
        return $this->belongsTo(User::class)->withDefault();
    }
}
```

Второй формат заносит в поля создаваемой записи значения из переданного в параметре *ассоциативного массива*:

```
public function user() {
    return $this->belongsTo(User::class)
                ->withDefault(['name' => 'guest']);
}
```

Третий формат позволяет занести значения в запись-заглушку программно. Заданная *анонимная функция* должна принимать два параметра: объект связанной модели, представляющей запись-заглушку, и объект текущей модели (именно в таком порядке!) — и заносить нужные значения в поля записи-заглушки. Пример:

```
public function user() {
    return $this->belongsTo(User::class)
                ->withDefault(function ($user, $bb) {
                    $user->name = 'guest.' . config('app.name');
                });
}
```

5.4.9. Замкнутая связь

Laravel позволяет создавать *замкнутые связи*, в которых таблица фактически связана сама с собой.

Предположим, что нужно организовать на доске объявлений двухуровневую систему рубрик: рубрика первого уровня «Недвижимость» содержит рубрики второго уровня «Дома», «Гаражи» и «Дачи», рубрика первого уровня «Автомобили» — рубрики второго уровня «Легковые» и «Грузовые», и т. п.

Рубрики обоих уровней мы будем хранить в таблице `rubrics`. Поле `parent_id` этой таблицы, необязательное к заполнению, будет хранить ключ рубрики первого уровня, в которую вложена текущая рубрика второго уровня. У рубрик первого уровня поле `parent_id` будет пустым.

Пример кода миграции, создающего поле `parent_id`:

```
Schema::create('rubrics', function (Blueprint $table) {
    . . .
    $table->unsignedBigInteger('parent_id')->nullable();
    $table->foreign('parent_id')->references('id')
          ->on('rubrics')->onDelete('restrict');
    . . .
});
```

Код модели `Rubric`, создающий замкнутую связь типа «один-со-многими»:

```
class Rubric extends Model {
    public function rubrics() {
        return $this->hasMany(self::class, 'parent_id');
    }
```

```
    public function parent() {
        return $this->belongsTo(self::class, 'parent_id');
    }
}
```

5.5. Методы моделей

В классе модели можно объявлять методы, реализующие какую-либо функциональность. Пример метода, выдающего строку с именем и адресом пользователя:

```
class User extends Authenticatable {
    public function getNameAndEmail() {
        return $this->name . ' (' . $this->email . ')';
    }
}
```

Его можно вызвать у любого объекта модели, в которой он объявлен:

```
$user = User::find(1);
$nameEmail = $user->getNameAndEmail();
```

Выполнить какие-либо дополнительные действия перед и (или) после сохранения записи можно, переопределив унаследованный от суперкласса метод `save()`:

```
class Rubric extends Model {
    . . .
    public function save(array $options = []) {
        // Выполняем какие-либо действия перед сохранением записи
        parent::save($options);
        // Выполняем что-либо после сохранения
    }
}
```

Чтобы выполнить какие-либо дополнительные действия перед и (или) после удаления записи, достаточно переопределить метод `delete()`:

```
class Rubric extends Model {
    . . .
    public function delete() {
        // Выполняем какие-либо действия перед удалением записи
        parent::delete();
        // Выполняем что-либо после удаления
    }
}
```

5.6. Преобразователи. Акцессоры и мутаторы

Когда внешний по отношению к модели код (например, контроллер) запрашивает у модели значение какого-либо поля, модель извлекает из поля значение, преобразует его к типу, заданному в свойстве `casts` модели (см. *разд. 5.3.3*), и выдает запросившему его коду. Когда внешний код заносит в поле модели новое значение, модель преобразует его к типу, поддерживаемому базой данных, и заносит в поле.

Однако часто бывает необходимо выполнять какие-либо дополнительные преобразования значения после его извлечения из поля и перед выдачей внешнему коду и (или), наоборот, после получения от внешнего кода и перед занесением в поле. Например, в имени зарегистрированного пользователя после извлечения его из поля приводить первую букву к верхнему регистру, а перед записью в поле — приводить все буквы к нижнему регистру.

Реализовать подобного рода действия можно непосредственно в классе модели, с помощью защищенного метода, называемого *преобразователем*.

Имя преобразователя должно совпадать с именем поля, значение которого следует преобразовать, набранным со строчной буквы. Если имя поля состоит из нескольких слов, разделенных символами подчеркивания, то в имени акцессора второе и последующие слова следует набрать вплотную, с прописных букв и без подчеркиваний. Такой стиль именования называется *camelCase*. Примеры имен полей и соответствующих им преобразователей: `title` — `title`, `file_name` — `fileName`.

В качестве типа возвращаемого преобразователем результата следует указать класс `Illuminate\Database\Eloquent\Casts\Attribute`.

Акцессор должен возвращать объект класса `Attribute`. Создавать его следует с помощью статического метода `make()` этого класса:

```
make([get=<акцессор>][, ][set=<мутатор>])
```

Метод принимает два параметра, которые рекомендуется указывать как именованные[2] (чтобы сделать код более удобочитаемым):

- `get` — *акцессор* — анонимная функция, преобразующая значение после извлечения его из поля и перед выдачей внешнему коду.

  Акцессор должен принимать в качестве параметра значение, извлеченное из поля и подлежащее преобразованию, и возвращать в качестве результата преобразованное значение.

  Если акцессор не указан, значение после извлечения из поля не будет подвергаться никаким дополнительным преобразованиям;

- `set` — *мутатор* — анонимная функция, преобразующая значение после получения его от внешнего кода и перед занесением в поле.

  Мутатор должен принимать в качестве параметра значение, полученное от внешнего кода и подлежащее преобразованию, и возвращать в качестве результата преобразованное значение.

  Если мутатор не указан, значение перед занесением в поле не будет подвергаться никаким дополнительным преобразованиям.

Пример преобразователя, объявленного в модели `User`, который перед выдачей имени пользователя приводит его первую букву к верхнему регистру, а перед записью в поле — приводит все буквы к нижнему регистру:

---

[2] Поддержка именованных параметров в функциях и методах появилась в PHP 8.0.

```
use Illuminate\Database\Eloquent\Casts\Attribute;
class User extends Authenticatable {
    . . .
    protected function name(): Attribute {
        return Attribute::make(
            get: fn ($value) => ucfirst($value),
            set: fn ($value) => strtolower($value)
        );
    }
}
```

Акцессор, заданный у поля, выполняется при каждом обращении к этому полю. Однако можно предписать Laravel выполнять акцессор только при первом обращении к полю и сохранять преобразованное значение в памяти, а при последующих обращениях к полю выдавать ранее сохраненное значение (это может пригодиться, если в акцессоре выполняются какие-либо длительные вычисления). Чтобы сделать это, достаточно вызвать у созданного объекта класса `Attribute` метод `shouldCache()`, например:

```
return Attribute::make( ... )->shouldCache();
```

5.6.1. Виртуальные поля

Виртуальным называется вычисляемое поле, значение которого рассчитывается не СУБД (как в случае обычных вычисляемых полей), а фреймворком. Виртуальное поле реализуется в виде преобразователя, содержащего только акцессор.

Пример создания в модели `Rubric` виртуального поля `full_name`, которое у рубрик первого уровня возвращает название, а у рубрик второго уровня — комбинацию названия родительской рубрики первого уровня и текущей рубрики:

```
class Rubric extends Model {
    . . .
    protected function fullName(): Attribute {
        return Attribute::make(
            get: fn ($value) =>
                     ($this->parent) ?
                         $this->parent->name . ' - ' . $this->name
                         :
                         $this->name
        );
    }
}
```

Пример обращения к этому виртуальному полю:

```
$rubric = Rubric::find(2);
$rubricFullName = $rubric->full_name;
```

5.6.2. Акцессоры и мутаторы в предыдущих версиях Laravel

В предыдущих версиях Laravel преобразователи не поддерживались, а акцессоры и мутаторы объявлялись в виде отдельных общедоступных методов:

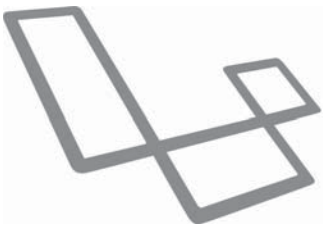
- акцессор — должен иметь имя формата `get<`*`имя поля в PascalCase`*`>Attribute`, принимать с единственным параметром извлеченное из поля значение, подлежащее преобразованию, и возвращать преобразованное значение;
- мутатор — должен иметь имя формата `set<`*`имя поля в PascalCase`*`>Attribute`, принимать с единственным параметром полученное от внешнего кода значение, подлежащее преобразованию, и после преобразования заносить его в поле, присвоив соответствующему элементу ассоциативного массива, хранящегося в свойстве `attributes` модели.

Этот синтаксис объявления акцессоров и мутаторов поддерживается и в Laravel 9.

Пример объявления акцессора и мутатора, которые выполняют преобразование имени пользователя в модели `User`, аналогичное реализованному в *разд. 5.6*:

```
class User extends Model {
    . . .
    public function getNameAttribute($value) {
        return ucfirst($value);
    }

    public function setNameAttribute($value) {
        $this->attributes['name'] = strtolower($value);
    }
}
```



ГЛАВА 6

Запись данных

6.1. Добавление, правка и удаление записей с помощью моделей

Удобнее всего добавлять, править и удалять записи таблицы посредством обслуживающей ее модели. Помимо этого, при использовании моделей:

- заполняются поля отметок создания и правки;
- выполняются методы `save()` и `delete()`. Если эти методы были переопределены с целью добавить дополнительную функциональность, последняя будет задействована;
- генерируются события модели (разговор о них пойдет в *главе 22*).

6.1.1. Добавление записей. Построитель запросов

Добавить в таблицу новую запись можно тремя способами.

Первый способ — создание «пустой» записи, занесение значений в отдельные поля и сохранение записи — выполняется в три этапа:

1. Создание объекта модели, представляющего «пустую» запись, — вызовом конструктора класса модели без параметров:

```
>>> use App\Models\Rubric;
>>> $rubric = new Rubric();
```

2. Занесение значений в поля записи — путем присваивания их одноименным свойствам объекта модели, созданного ранее:

```
>>> // Создаем рубрику первого уровня «Недвижимость»
>>> // (используется класс модели с замкнутой связью из разд. 5.4.8)
>>> $rubric->name = 'Недвижимость';
>>> $rubric->parent_id = null;
```

3. Сохранение записи — вызовом метода `save([<массив настроек>])` объекта модели:

```
>>> $rubric->save();
```

Метод возвращает `true`, если запись была успешно сохранена, и `false` — в противном случае.

В переданном ассоциативном *массиве* можно указать настройки сохранения записи. В настоящее время поддерживается только настройка `touch`, при значении `true` указывающая модели занести значения в поля отметок создания и правки (поведение по умолчанию), а при значении `false` — не заносить значения в эти поля. Пример:

```
>>> // Сохраняем запись, не занося значений в поля отметок создания и правки
>>> $rubric->save(['touch' => false]);
```

Второй способ — создание записи с одновременным заполнением значениями ее полей и сохранение записи — выполняется в два этапа:

1. Создание объекта модели, представляющего запись, — вызовом конструктора класса модели в формате:

   ```
   new <модель>(<ассоциативный массив со значениями полей>)
   ```

   Ключи элементов *ассоциативного массива* должны совпадать с именами полей, а значения элементов зададут значения для этих полей:

   ```
   >>> // Создаем рубрику второго уровня «Дома»,
   >>> // вложенную в рубрику «Недвижимость»
   >>> $rubric2 = new Rubric(['name' => 'Дома',
   ...                        'parent_id' => $rubric->id]);
   ```

   > ***ИСПОЛЬЗУЕТСЯ МАССОВОЕ ПРИСВАИВАНИЕ!***
   >
   > Не забываем, что при массовом присваивании значения заносятся только в поля, присутствующие в массиве из свойства `fillable` модели, или, наоборот, *не* присутствующие в массиве из свойства `guarded`.

2. Сохранение записи — вызовом того же метода `save()` объекта модели:

   ```
   >>> $rubric2->save();
   ```

Третий способ — создание, заполнение путем массового присваивания и сохранение записи за один этап — выполняется вызовом метода `create()`:

```
create(<ассоциативный массив со значениями полей>)
```

В качестве результата возвращается объект созданной и уже сохраненной записи.

Этот метод можно вызвать как обычный — у любого объекта той модели, в которую нужно добавить запись:

```
>>> // Создаем еще одну рубрику второго уровня «Гаражи»,
>>> // вложенную в рубрику «Недвижимость»
>>> $rubric3 = $rubric2->create(['name' => 'Гаражи',
...                              'parent_id' => $rubric->id]);
```

Также метод `create()` можно вызвать у класса модели — как статический:

```
>>> // Создаем объявление о продаже гаража
>>> use App\Models\User;
>>> use App\Models\Bb;
>>> $user = User::first();
```

```
>>> $bb = Bb::create(['title' => 'Гараж', 'content' => 'На две машины',
...                   'address' => 'Проезд № 3', 'price' => 300000,
...                   'rubric_id' => $rubric3->id,
...                   'user_id' => $user->id]);
```

Метод `create()`, как и ряд других, хотя и вызывается у модели, но выполняется *построителем запросов* — подсистемой, формирующей SQL-запросы, отправляющей их базе данных и получающей результат их выполнения. При попытке вызвать метод построителя запросов у модели последняя создает новый объект построителя запросов и передает вызов метода ему.

> ***МОДЕЛИ ТОЖЕ ИСПОЛЬЗУЮТ ПОСТРОИТЕЛЬ ЗАПРОСОВ...***
>
> ...«за кулисами» — для формирования окончательного SQL-запроса, пересылаемого базе данных.
>
> Методы построителя запросов выполняются быстрее, но аналогичные инструменты моделей обеспечивают бóльшую гибкость.

Следующие два метода также выполняются построителем запросов и поэтому работают быстро:

- `firstOrNew()` — ищет запись, чьи поля содержат значения из первого *массива*, и возвращает хранящий ее объект модели. Если подходящая запись не найдена, создает новую запись, занося в ее поля значения из обоих *массивов* путем массового присваивания, и возвращает в качестве результата. Созданную запись *не* сохраняет в базе данных. Формат вызова:

  ```
  firstOrNew(<массив с искомыми значениями>[,
             <массив со значениями, заносимыми в создаваемую запись>])
  ```

  Оба *массива* должны быть ассоциативными. Ключи их элементов соответствуют полям таблицы, а значения элементов — значениям этих полей. Пример:

  ```
  >>> // Ищем рубрику первого уровня «Автомобили» и, если не найдем,
  >>> // создаем ее
  >>> $rubric3 = Rubric::firstOrNew(['name' => 'Автомобили',
  ...                                'parent_id' => null]);
  >>> $rubric3->save();
  ```

- `firstOrCreate()` — то же самое, что и `firstOrNew()`, только дополнительно сохраняет созданную запись:

  ```
  >>> // Ищем рубрику «Легковые» и, если не найдем, создаем ее как
  >>> // рубрику второго уровня, подчиненную рубрике «Автомобили»
  >>> $rubric4 = Rubric::firstOrCreate(['name' => 'Легковые'],
  ...                                  ['parent_id' => $rubric3->id]);
  >>> // Создаем еще одну рубрику
  >>> $rubric5 = Rubric::firstOrCreate(['name' => 'Компрессоры']);
  ```

6.1.2. Правка записей

Правка записи выполняется в три этапа:

1. Получение объекта модели, представляющего запись, — любым из способов, рассмотренных в *главах 1* и *7*:

```
>>> // Извлекаем нужную рубрику
>>> $rubric = Rubric::find(1);
>>> echo $rubric->name;
Недвижимость
```

2. Занесение в поля записи новых значений — одним из следующих способов:
   - занесением значений в отдельные свойства модели:

     ```
     >>> $rubric->name = 'Здания';
     >>> $rubric->parent_id = null;
     ```

   - массовым присваиванием новых значений нескольким полям одновременно — вызовом метода `fill()` модели:

     `fill(<`*ассоциативный массив с заносимыми значениями*`>)`

     Пример:

     ```
     >>> $rubric->fill(['name' => 'Здания', 'parent_id' => null]);
     ```

3. Сохранение записи — вызовом метода `save()`:

   ```
   >>> $rubric->save();
   ```

Метод `update()` модели заменяет вызовы методов `fill()` и `save()`, имеет тот же формат вызова, что и метод `fill()`, и возвращает `true`, если запись была успешно сохранена, и `false` — в противном случае:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Автомобили']);
>>> $rubric->update(['name' => 'Транспорт']);
```

Метод `updateOrCreate()` построителя запросов ищет запись, чьи поля содержат значения из первого *массива*, заносит в нее значения из второго *массива*, сохраняет и возвращает в качестве результата. Если подходящая запись не найдена, метод создаст новую запись, занесет в ее поля значения из обоих *массивов*, сохранит и также вернет. Формат вызова:

```
updateOrCreate(<массив с искомыми значениями>[,
               <массив со значениями, заносимыми в запись>])
```

Оба *массива* должны быть ассоциативными. Ключи их элементов соответствуют полям таблицы, а значения элементов — значениям этих полей. Пример:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Транспорт']);
>>> // Поскольку рубрика «Грузовой» отсутствует, она будет создана и
>>> // станет рубрикой второго уровня, вложенной в рубрику «Транспорт»
>>> Rubric::updateOrCreate(['name' => 'Грузовой'],
...                        ['parent_id' => $rubric->id]);
```

6.1.2.1. Правка значений отдельных полей

Ряд методов, поддерживаемых моделью, позволяет исправить значения отдельных полей записи:

- `increment()` — увеличивает значение указанного числового *поля* текущей записи на заданную *величину* (по умолчанию: 1) и сохраняет запись:

```
increment(<имя поля>[, <величина>=1[,
        <массив с новыми значениями других полей>]])
```

Можно указать ассоциативный *массив* со значениями, которые будут занесены в другие поля записи. Ключи его элементов должны совпадать с именами полей, а значения элементов зададут значения для этих полей. Пример:

```
>>> $bb->increment('price');
>>> echo $bb->price;
300001
>>> $bb->increment('price', 99, ['address' => 'Самый дальний гараж']);
>>> echo $bb->price;
300100
```

- `decrement()` — уменьшает значение указанного числового *поля* текущей записи на заданную *величину* (по умолчанию: 1) и сохраняет запись. Формат вызова такой же, как и у `increment()`;
- `touch([<имя поля>=null])` — заносит текущую временну́ю отметку в поле отметки правки и сохраняет запись. Если в параметре передать *имя* какого-либо поля, временна́я отметка будет занесена в это поле.

6.1.2.2. Проверка, значения каких полей изменились

Выяснить, в какие поля были занесены новые значения после извлечения записи, но перед ее сохранением, позволят два следующих метода модели:

- `isDirty([<имя поля>|<массив с именами полей>])` — в зависимости от использованного формата вызова:
  - `isDirty()` (без параметров) — возвращает `true` в случае, если хотя бы одно поле текущей записи получило новое значение, и `false` — в противном случае;
  - `isDirty(<имя поля>)` — возвращает `true`, если поле с указанным *именем* получило новое значение, и `false` — в противном случае;
  - `isDirty(<имя поля 1>, <имя поля 2>, ... <имя поля n>)`

    или

    `isDirty(<массив с именами полей>)` — возвращает `true`, если хотя бы одно поле с именем, указанным в параметрах или в переданном *массиве*, получило новое значение, и `false` — в противном случае.

    Пример:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Легковые']);
>>> $rubric->isDirty();
=> false
>>> $rubric->isDirty('name');
=> false
>>> $rubric->isDirty('parent_id');
=> false
>>> $rubric->name = 'Легковой';
>>> $rubric->isDirty();
=> true
```

```
>>> $rubric->isDirty('name');
=> true
>>> $rubric->isDirty('parent_id');
=> false
>>> $rubric->isDirty('name', 'parent_id');
=> true
>>> $rubric->isDirty(['name', 'parent_id']);
=> true
>>> $rubric->save();
>>> $rubric->isDirty();
=> false
>>> $rubric->isDirty('name');
=> false
>>> $rubric->isDirty('parent_id');
=> false
```

- `isClean()` — противоположен методу `isDirty()`, т. е. возвращает `false`, если значения в полях записи изменились, и `true` — в противном случае.

Метод `wasChanged()` модели поддерживает те же форматы вызова, что и метод `isDirty()`, и определяет, изменились ли значения в полях записи после ее сохранения во время обработки текущего клиентского запроса. Если значения изменились, возвращается `true`, иначе — `false`.

Метод `getOriginal([<имя поля>])` — при вызове:

- без параметров — возвращает ассоциативный массив с изначальными значениями полей текущей записи. Ключи возвращаемого массива соответствуют полям, а значения — значениям этих полей;
- с *именем поля* в качестве параметра — возвращает изначальное значение поля с указанным *именем*.

Пример:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Легковые']);
>>> $rubric->wasChanged();
=> false
>>> $rubric->wasChanged('name');
=> false
>>> $rubric->wasChanged('parent_id');
=> false
>>> $rubric->wasChanged('name', 'parent_id');
=> false
>>> $rubric->fill(['name' => 'Легковой']);
>>> $rubric->getOriginal('name');
=> "Легковые"
>>> $rubric->save();
>>> $rubric->wasChanged();
=> true
>>> $rubric->wasChanged('name');
=> true
```

```
>>> $rubric->wasChanged('parent_id');
=> false
>>> $rubric->wasChanged(['name', 'parent_id']);
=> true
```

6.1.3. Удаление записей

Удаление записи выполняется в два этапа:

1. Получение объекта модели:

   ```
   >>> $rubric = Rubric::firstOrNew(['name' => 'Компрессоры']);
   ```

2. Удаление записи — вызовом метода `delete()` у ее объекта:

   ```
   >>> $rubric->delete();
   ```

   Метод возвращает `true`, если запись была успешно удалена, и `false` — в противном случае.

Для удаления сразу нескольких записей удобно применять статический метод `destroy()` модели. Форматы вызова:

```
destroy(<ключ записи 1>, <ключ записи 2>, ... <ключ записи n>)
destroy(<массив или коллекция ключей записей>)
```

В качестве результата возвращается количество удаленных записей. Примеры:

```
use App\Models\Bb;
Bb::destroy(10, 20, 35);
Bb::destroy([10, 20, 35])
// Здесь методу передается коллекция Laravel (подробности — в главе 15)
Bb::destroy(collect([10, 20, 35]))
```

> ***КАСКАДНОЕ УДАЛЕНИЕ СВЯЗАННЫХ ЗАПИСЕЙ ВЫПОЛНЯЕТСЯ НА УРОВНЕ СУБД, А НЕ МОДЕЛИ!***
>
> Соответственно при этом не выполняется метод `delete()` модели (который может быть переопределен с целью выполнения дополнительных действий) и не генерируются события. Если необходимо, чтобы этот метод выполнялся, а события генерировались, следует перебрать все связанные записи и удалить каждую из них принудительно. Пример:
>
> ```
> // Принудительно удаляем все объявления, связанные с рубрикой из переменной
> // rubric
> foreach ($rubric->bbs()->get() as bb) {
>     $bb->delete();
> }
> // После чего удаляем саму рубрику
> $rubric->delete();
> ```

6.1.3.1. «Мягкое» удаление записей

Если и модель, и обслуживаемая ею таблица поддерживают «мягкое» удаление (см. *разд. 4.1.3.2* и *5.3.4*), при вызове метода `delete()` у записи последняя подвергнется «мягкому» удалению. Пример:

```
// Модель Rubric поддерживает «мягкое» удаление
$rubric = Rubric::firstOrNew(['name' => 'Легковой']);
// Будет выполнено «мягкое» удаление рубрики «Легковой»
$rubric->delete();
```

Восстановить «мягко» удаленную запись можно вызовом у нее метода `restore()`:

```
$rubric->restore();
```

Он возвращает `true`, если запись была успешно восстановлена, и `false` — в противном случае.

Метод `trashed()` модели возвращает `true`, если текущая запись была «мягко» удалена, и `false` — в противном случае:

```
if ($rubric->trashed()) {
    // Запись была «мягко» удалена
}
```

Чтобы полностью удалить запись модели, поддерживающей «мягкое» удаление, следует вызвать у нее метод `forceDelete()`:

```
$rubric->forceDelete();
```

6.1.4. Работа со связанными записями

6.1.4.1. Связи «один-со-многими» и «один-с-одним»: связывание существующих записей

Чтобы связать уже существующие записи таблиц, между которыми установлена связь «один-со-многими» или «один-с-одним», следует:

- у записи вторичной таблицы — выполнить одно из двух:
  - занести в поле внешнего ключа ключ связываемой записи первичной таблицы:

    ```
    >>> $rubric = Rubric::firstOrNew(['name' => 'Здания']);
    >>> $rubric2 = Rubric::create(['name' => 'Дачи',
    ...                            'parent_id' => $rubric->id]);
    ```

  - вызвать метод, создающий «обратную» связь, и вызвать у возвращенного им объекта «обратной» связи метод `associate()`:

    `associate(<`*связываемая запись первичной таблицы*`>)`

    Метод возвращает текущую запись в качестве результата и не сохраняет ее, поэтому далее следует сохранить запись явно. Пример:

    ```
    >>> // Создаем рубрику второго уровня «Прочие» и связываем ее с
    >>> // рубрикой первого уровня «Здания»
    >>> $rubric3 = new Rubric(['name' => 'Прочее']);
    >>> $rubric3->parent()->associate($rubric);
    >>> $rubric3->save();
    ```

- у записи первичной таблицы — вызвать метод модели, создающий «прямую» связь, и вызвать у возвращенного им объекта «прямой» связи один из следующих методов:
  - `save(<`*вновь созданная запись*`>)` — связывает переданную ему *вновь созданную запись* вторичной модели и сразу же сохраняет ее. Метод возвращает переданную

ему *запись*, если связывание и сохранение прошли успешно, или `false` — в противном случае. Пример:

```
>>> $bb = new Bb(['title' => 'Дача', 'content' => 'Новая',
...               'address' => 'Проезд № 4', 'price' => 100000,
...               'user_id' => $user->id]);
>>> $rubric2->bbs()->save($bb);
```

- `saveMany(`*<массив вновь созданных записей>*`)` — связывает все записи, содержащиеся в переданном ему *массиве*, который и возвращает в качестве результата:

```
>>> $rubric2->bbs()->saveMany([new Bb( ... ),
...                            new Bb( ... )]);
```

Если поле внешнего ключа во вторичной таблице необязательно к заполнению (может хранить значения `null`), можно разорвать связь между записями. Для этого следует в записи вторичной таблицы выполнить одно из следующих действий:

- присвоить полю внешнего ключа значение `null` и сохранить запись;

```
>>> $rubric2->parent_id = null;
>>> $rubric2->save();
```

- вызвать метод, создающий «обратную» связь, и вызвать у возвращенного им объекта «обратной» связи метод `dissociate()`, после чего сохранить запись вторичной модели. Метод `dissociate()` возвращает текущую запись вторичной таблицы. Пример:

```
>>> $rubric2->parent()->dissociate()->save();
```

6.1.4.2. Связи «один-со-многими» и «один-с-одним»: добавление и правка связанных записей

При использовании связей «один-со-многими» и «один-с-одним» для создания новых записей во вторичной таблице с одновременным связыванием их с текущей записью первичной таблицы объект «прямой» связи предоставляет методы:

- `create()` — полностью аналогичный описанному в *разд. 6.1.1*:

```
>>> // Создаем дополнительные сведения о пользователе и связываем
>>> // их с первым пользователем (модели User и Account,
>>> // связанные связью «один-с-одним», были объявлены в разд. 5.4.3)
>>> use App\Models\User;
>>> $user = User::first();
>>> $user->account()->create(['first_name' => 'Иван',
...                           'last_name' => 'Иванов']);
```

- `createMany(`*<массив массивов со значениями полей>*`)` — создает новые записи, заносит в поля каждой из них значения из очередного ассоциативного массива, содержащегося в переданном *массиве*, связывает созданные записи с текущей, сохраняет их и возвращает в качестве результата переданный *массив*:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Дома']);
>>> $bb_a1 = ['title' => 'Дом', 'content' => 'Большой, трехэтажный',
...           'address' => 'Главная ул.', 'price' => 5000000,
...           'user_id' => $user->id];
```

```
>>> $bb_a2 = ['title' => 'Дом', 'content' => 'Маленький, фанерный',
...           'address' => 'Окраинный пер.', 'price' => 200000,
...           'user_id' => $user->id];
>>> $rubric->bbs()->createMany([$bb_a1, $bb_a2]);
```

Объект «прямой» связи также поддерживает методы: `firstOrNew()`, `firstOrCreate()` и `updateOrCreate()`, описанные в *разд. 6.1.1* и *6.1.2*.

Исправить значения какого-либо поля связанной первичной записи из вторичной записи можно, обратившись к свойству объекта вторичной записи, чье имя совпадает с именем метода, создающего «обратную» связь (например, если «обратную» связь создает метод `rubric()`, нужно обратиться к свойству `rubric`). Это свойство хранит объект модели, представляющий связанную запись первичной таблицы. Пример:

```
>>> // К какой рубрике относится объявление из переменной bb?
>>> echo $bb->rubric->name;
Дачи
>>> // Меняем название этой рубрики
>>> $bb->rubric->name = 'Частные дома';
```

Метод `save()` сохраняет только текущую запись. Чтобы сохранить одновременно и текущую, и связанные записи, следует вызвать метод `push()` модели:

```
>>> $bb->push();
```

Также можно исправить значение какого-либо поля любой связанной записи вторичной таблицы из записи первичной таблицы. Свойство объекта записи первичной таблицы, чье имя совпадает с именем метода, создающего «прямую» связь, хранит коллекцию связанных записей вторичной таблицы (например, если «прямую» связь создает метод `bbs()`, свойство называется `bbs`). Пример:

```
>>> // Какой адрес указан в первом объявлении из рубрики rubric?
>>> echo $rubric->bbs[1]->address;
Окраинный пер.
>>> // Исправляем его
>>> $rubric->bbs[0]->address = 'Овражный пр.';
>>> $rubric->push();
```

Для удаления связанных записей следует пользоваться инструментами, описанными в *разд. 6.1.3*.

6.1.4.3. Связь «один-с-одним из многих»

Для связывания записей таблиц, между которыми установлена связь «один-с-одним из многих» , можно использовать инструменты, описанные в *разд. 6.1.4.1* и *6.1.4.2*:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Легковой']);
>>> $rubric->latestBb()->create(['title' => 'Жигули',
...                              'content' => 'Совсем новые',
...                              'address' => 'Пр. Автовладельцев',
...                              'price' => 1000000,
...                              'user_id' => $user->id]);
```

6.1.4.4. Связь «многие-со-многими»: связывание записей

Для связывания существующих записей таблиц, между которыми установлена связь «многие-со-многими», следует в объекте одной из связываемых записей вызвать метод, создающий связь, а у возвращенного им объекта связи — один из следующих методов:

- `attach()` — связывает записи с заданными ключами с текущей записью. Форматы вызова:

```
attach(<ключ записи>[, <ассоциативный массив со значениями ↳
                        дополнительных полей связующей таблицы>])
attach(<массив с ключами записей>)
```

Первый формат вызова метода используется для связывания одной записи с заданным *ключом*. Методу можно передать *ассоциативный массив*, содержащий значения, которые будут занесены в дополнительные поля связующей таблицы.

Второй формат позволяет связать с текущей сразу несколько записей. Элемент передаваемого *массива* должен представлять собой либо ключ очередной связываемой записи, либо пару формата:

```
<ключ записи> => <ассоциативный массив со значениями полей связующей таблицы>
```

Примеры:

```
>>> use App\Models\Machine;
>>> use App\Models\Spare;
>>> $machine1 = Machine::create(['name' => 'Самосвал']);
>>> $spare1 = Spare::create(['name' => 'Заклепка']);
>>> $spare2 = Spare::create(['name' => 'Винт']);
>>> // Связываем первую машину (самосвал) с первой деталью
>>> // (заклепкой). Поскольку мы не указали количество, в поле cnt
>>> // связующей таблицы будет сохранено заданное для него
>>> // значение по умолчанию: 1 (см. разд. 5.4.4).
>>> $machine1->spares()->attach($spare1->id);
>>> // Помимо одной заклепки в самосвале будет 10 винтов
>>> $machine1->spares()->attach($spare2->id, ['cnt' => 10]);
>>> $machine2 = Machine::create(['name' => 'Тепловоз']);
>>> $machine3 = Machine::create(['name' => 'Будильник']);
>>> // Заклепки также будут входить в состав тепловоза
>>> // (100 штук) и будильника (1 штука — по умолчанию)
>>> $spare1->machines()->attach([$machine2->id => ['cnt' => 100],
...                              $machine3->id]);
```

- `sync()` — связывает с текущей записью записи с указанными в *массиве* ключами:

```
sync(<массив с ключами записей>[, <отсоединять записи?>=true])
```

Массив задается в том же формате, что и у метода `attach()`.

По умолчанию ранее связанные записи, если их ключи не присутствуют в *массиве*, будут отсоединены от текущей записи. Чтобы такие записи не отсоединялись, следует передать параметром *отсоединять записи* значение `false`.

Метод возвращает ассоциативный массив с ключами `attached` (количество вновь связанных записей), `detached` (количество отсоединенных записей) и `updated` (ко-

личество связанных записей, у которых были исправлены записи связующей таблицы).

Пример:

```
>>> $spare3 = Spare::create(['name' => 'Гайка']);
>>> // Теперь самосвал будет состоять из 10 винтов (их количество не
>>> // изменилось, поскольку мы не заносили новое значение в поле cnt
>>> // записи связующей таблицы) и 2 гаек. Заклепок в нем больше нет.
>>> $machine1->spares()->sync([$spare2->id,
...                            $spare3->id => ['cnt' => 2]]);
```

- `syncWithoutDetaching(<массив с ключами записей>)` — то же самое, что и `sync()`, только не отсоединяет связанные ранее записи, чьи ключи не были указаны в заданном *массиве*:

```
>>> // Добавляем в самосвал три заклепки
>>> $machine1->spares()
...          ->syncWithoutDetaching([$spare1->id => ['cnt' => 20]]);
```

- `syncWithPivotValues()` — связывает с текущей записью записи с указанными в первом *массиве* ключами и заносит в дополнительные поля всех записей связующих таблицы, созданных в результате связывания, значения из заданного второго *массива*:

```
syncWithPivotValues(<массив ключей записей>,
                    <ассоциативный массив со значениями ↪
                    дополнительных полей связующей таблицы>[,
                    <отсоединять записи?>=true])
```

Назначение параметра *отсоединять записи* такое же, как и у метода `sync()`, и таков же возвращаемый результат. Пример:

```
>>> $machine4 = Machine::create(['name' => 'Трактор']);
>>> $machine4->spares()->syncWithPivotValues([$spare1->id,
...                                           $spare2->id],
...                                          ['cnt' => 4]);
```

- `toggle(<массив с ключами записей>)` — перебирает заданный *массив с ключами записей* и проверяет, связана ли запись с очередным указанным в *массиве* ключом с текущей записью. Если запись связана, он отсоединяет ее, если не связана — связывает. Возвращает тот же результат, что и метод `sync()`. Пример:

```
>>> $spare4 = Spare::create(['name' => 'Штифт']);
>>> $spare4->machines()->toggle([$machine2->id, $machine4->id]);
```

Объект связи «многие-со-многими» также поддерживает методы `save()` и `saveMany()`, описанные в *разд. 6.1.4.1*. Во втором параметре этим методам можно передать:

- методу `save()` — ассоциативный массив со значениями полей связующей таблицы (если в эти поля нужно занести новые значения);

```
>>> $spare5 = new Spare(['name' => 'Шайба']);
>>> $machine3->spares()->save($spare5, ['cnt' => 200]);
```

- методу `saveMany()` — массив с ассоциативными массивами, содержащими значения для полей связующей таблицы:

```
>>> $spare5->machines()
...         ->saveMany([new Machine(['name' => 'Керогаз']),
...                     new Machine(['name' => 'Велосипед'])],
...                    [['cnt' => 1], ['cnt' => 5]]);
```

6.1.4.5. Связь «многие-со-многими»: добавление и правка связанных записей

Объект связи «многие-со-многими» поддерживает методы `create()` и `createMany()`, описанные в *разд. 6.1.4.2*. Вторым параметром им можно передать:

- ❐ методу `create()` — ассоциативный массив со значениями полей связующей таблицы:

```
>>> echo $spare5->machines[2]->name;
Велосипед
>>> $spare5->machines[2]->spares()
...                     ->create(['name' => 'Колесо'], ['cnt' => 2]);
```

- ❐ методу `createMany()` — массив с ассоциативными массивами, содержащими значения для полей связующей таблицы.

Исправить значение поля связанной записи можно способом, описанным в *разд. 6.1.4.2*:

```
>>> echo $spare5->machines[2]->name;
Велосипед
>>> $spare5->machines[2]->name = 'Мотоцикл';
>>> $spare5->push();
```

Исправить значение поля в записи связующей таблицы можно посредством свойства `pivot` (это имя по умолчанию, изменить его можно вызовом метода `as()` объекта связи — см. *разд. 5.4.4*), принадлежащего объекту связанной записи. Оно хранит объект либо автоматически сгенерированной модели, либо связующей модели (если таковая была создана явно — см. *разд. 5.4.4.1*). Пример:

```
>>> // Сколько в самосвале (переменная machine1) винтов
>>> // (первая из связанных деталей)?
>>> echo $machine1->spares[0]->pivot->cnt;
10
>>> // Пусть будет 9
>>> $machine1->spares[0]->pivot->cnt = 9;
>>> $machine1->spares[0]->pivot->save();
```

Править значения в полях записи связующей таблицы также можно методом `updateExistingPivot()`, вызываемым у объекта связи:

```
updateExistingPivot(<ключ связанной записи>,
      <ассоциативный массив со значениями полей связующей таблицы>)
```

Пример:

```
>>> // Пусть в самосвале будет 9 гаек (переменная spare3)
>>> $machine1->spares()->updateExistingPivot($spare3->id, ['cnt' => 9]);
```

Объект связи «многие-со-многими» также поддерживает методы: `firstOrNew()`, `firstOrCreate()` и `updateOrCreate()`, описанные в *разд. 6.1.1* и *6.1.2*.

6.1.5. Копирование записей

Если нужно добавить в таблицу несколько записей с примерно одинаковым содержанием, удобно создать одну запись, а потом сделать необходимое количество ее копий, произведя в них нужные изменения. Копирование записи выполняет метод `replicate()` модели:

```
replicate([<массив с именами полей, игнорируемых при копировании>])
```

В создаваемую копию записи копируются значения всех полей текущей записи, кроме ключевого, отметок создания, правки и удаления (если оно присутствует). Можно указать *массив* с именами полей, значения которых также не будут копироваться в создаваемую запись. Метод возвращает несохраненную копию текущей записи. Пример:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Грузовой']);
>>> // Создаем исходное объявление
>>> $bb = Bb::create(['title' => 'ЗИЛ', 'content' => 'Старый, ржавый',
...                   'address' => 'Писать мне', 'price' => 1000000,
...                   'rubric_id' => $rubric->id,
...                   'user_id' => $user->id]);
>>> // Делаем его копию
>>> $bb2 = $bb->replicate(['content', 'price']);
>>> // Заносим новые значения в поля копии
>>> $bb2->fill(['content' => 'Новый', 'price' => 10000000]);
>>> // И сохраняем ее
>>> $bb2->save();
```

6.2. Массовые добавление, правка и удаление записей

Для добавления, правки и удаления большого количества записей лучше пользоваться не моделями, а непосредственно построителем запросов — для повышения быстродействия. Его методы, как говорилось ранее, вызываются либо у класса модели (как статические), либо у объекта модели (как обычные) — в этом случае модель создаст новый объект построителя запросов и передаст вызов метода ему.

Однако при использовании построителя запросов:

- поля отметок создания и правки — не заполняются и не изменяются;
- методы `save()` и `delete()` — не выполняются. Соответственно не будет работать дополнительная функциональность, записанная в этих методах при их переопределении;
- события моделей — не генерируются.

6.2.1. Массовое добавление записей

Для массового добавления записей применяются методы:

- `insert()` — добавляет запись в таблицу. Форматы вызова:

```
insert(<ассоциативный массив со значениями полей>)
insert(<массив, содержащий ассоциативные массивы со значениями полей>)
```

Первый формат добавляет одну запись, второй — несколько. Метод возвращает `true`, если добавление записей прошло успешно, и `false` — в противном случае. Примеры:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Легковой']);
>>> BB::insert(['title' => 'Запорожец',
...             'content' => 'Старый, ржавый, сильно битый',
...             'address' => 'На помойке', 'price' => 10000,
...             'rubric_id' => $rubric->id, 'user_id' => $user->id]);

>>> $rubric2 = Rubric::firstOrNew(['name' => 'Грузовой']);
>>> // Добавляем сразу две записи
>>> BB::insert([['title' => 'МАЗ', 'content' => 'Старый, заслуженный',
...              'address' => 'На стоянке', 'price' => 4000000,
...              'rubric_id' => $rubric2->id, 'user_id' => $user->id],
...             ['title' => 'ГАЗ', 'content' => 'Совсем новый',
...              'address' => 'На стоянке', 'price' => 70000000,
...              'rubric_id' => $rubric2->id,
...              'user_id' => $user->id]]);
```

При попытке добавить записи, содержащие повторяющиеся значения в уникальном поле, возникнет ошибка;

- `insertOrIgnore()` — то же самое, что и `insert()`, только при попытке добавить записи, содержащие повторяющиеся значения в уникальном поле, ошибки не возникнет, и в качестве результата возвращается количество добавленных записей;
- `insertUsing()` — добавляет записи, извлеченные указанным *вложенным запросом*:

  ```
  insertUsing(<массив с именами полей>, <вложенный запрос>)
  ```

  Вложенный запрос формируется способом, описанным в *главе 7*. Он должен выдавать те же поля и том же порядке, что приведены в заданном *массиве*. Возвращаемый результат — количество добавленных записей. Пример:

  ```
  // Переносим объявления, созданные более 10 лет назад, в модель OldBb
  OldBb::insertUsing(
    ['title', 'content', 'address', 'price', 'rubric_id', 'user_id'],
    Bb::select('title', 'content', 'address', 'price', 'rubric_id',
               'user_id')
      ->where('created_at', '<', now()->subYears(10))
  );
  ```

- `insertGetId()` — добавляет запись и возвращает ее автоматически сгенерированный ключ:

  ```
  insertGetId(<ассоциативный массив со значениями полей>[,
              <имя автоматического счетчика>])
  ```

  Во втором параметре, используемом только PostgreSQL, указывается *имя автоматического счетчика*, из которого берется значение генерируемого ключа, если этот счетчик имеет имя, отличное от `id`.

6.2.2. Массовая правка записей

Для массовой правки записей построитель запросов предоставляет метод `update()`, аналогичный рассмотренному в *разд. 6.1.2*, за тем исключением, что он возвращает количество исправленных записей;

Если этот метод вызвать без указаний условий фильтрации:

```
>>> Bb::update(['publish' => true]);
```

он исправит все записи, что есть в таблице. Чтобы исправить отдельные записи, их предварительно нужно отфильтровать — например, вызовом метода `where()` (был кратко описан в *разд. 1.10*):

```
>>> Bb::where('publish', false)->update(['publish' => true]);
```

При правке значений полей типа `JSON` для доступа к свойствам записанных в них объектов следует использовать «стрелочный» синтаксис, применяемый в PHP:

```
>>> Bb::where('publish', false)->update(['state->reason' => 'Устарело']);
```

Метод `updateOrInsert()` полностью аналогичен методу `updateOrCreate()` (см. *разд. 6.1.2*), только в качестве результата возвращает текущий объект построителя запросов:

```
>>> use Illuminate\Support\Facades\Hash;
>>> User::updateOrInsert(['email' => 'editor@bboard.ru',
...                       'name' => 'editor'],
...                      ['password' => Hash::make('editor')]);
```

Метод `upsert()` исправляет записи модели, удовлетворяющие заданным условиям, или создает их, если подходящие записи не существуют. Формат вызова:

```
upsert(<массив массивов со всеми необходимыми значениями полей>,
       <массив с именами полей, по которым будет вестись поиск записей>[,
       <массив с именами полей, которые должны быть исправлены>])
```

Первый *массив* должен содержать ассоциативные массивы, каждый из которых задаст значения полей для одной из искомых записей — как для ее поиска, так и для ее исправления или создания. Ключи элементов вложенных ассоциативных массивов должны соответствовать полям модели, а значения элементов зададут значения полей.

Второй *массив* должен содержать имена полей, по которым будет вестись поиск записей (искомые значения этих полей Laravel возьмет из первого *массива*). По приведенным в нем полям должны быть созданы ключевые или уникальные индексы, иначе метод не сработает.

Третий *массив* включит имена полей, которые должны быть исправлены (новые значения, записываемые в эти поля, фреймворк также возьмет из первого *массива*). Если третий *массив* не указан, найденные записи исправляться не будут.

Если какая-либо запись не была найдена, она будет создана, и в ее поля будут записаны значения из первого *массива*.

Метод возвращает количество добавленных и исправленных записей.

Пример:

```
>>> // Пользователь editor с адресом editor@bboard.ru существует, и его
>>> // пароль будет изменен на supereditor. Пользователь trainee
>>> // с адресом trainee@bboard.ru не существует, и он будет создан.
>>> // Поле email — единственное в таблице users, по которому создан
>>> // уникальный индекс.
>>> User::upsert(
...     [
...         ['email' => 'editor@bboard.ru', 'name' => 'editor',
...          'password' => Hash::make('supereditor')],
...         ['email' => 'trainee@bboard.ru', 'name' => 'trainee',
...          'password' => Hash::make('12345')]
...     ],
...     ['email'], ['password']
... );
```

Построитель запросов также поддерживает методы `increment()` и `decrement()` (см. *разд. 6.1.2.1*), которые возвращают количество исправленных записей:

```
>>> Bb::where('title', 'Лопата')->increment('price', 50);
```

6.2.3. Массовое удаление записей

Массовое удаление записей выполняет метод `delete()`. В качестве результата он возвращает количество удаленных записей.

Будучи вызванным без предварительной фильтрации, метод удалит все записи. Для удаления отдельных записей их предварительно следует отфильтровать. Пример:

```
>>> Bb::where('publish', false)->delete();
```

Метод `truncate()` удаляет из таблицы все записи и сбрасывает счетчик автоинкремента в 0:

```
>>> Bb::truncate();
```

6.2.4. Использование фасада *DB* для записи данных

Объект построителя запросов можно получить посредством не только модели, но и фасада `Illuminate\Support\Facades\DB` (который скрывает за собой подсистему, обрабатывающую базы данных). Этот фасад пригодится, если необходимо занести в таблицу базы данных какие-либо записи, но модель, обслуживающая эту таблицу, еще не объявлена (такое может случиться, например, при программировании сидеров, описанных в *разд. 4.2*).

Однако, в отличие от модели, выдающей объект построителя запросов, который уже настроен на работу с нужной таблицей, построитель, выдаваемый фасадом `DB`, «пуст». Ему следует указать базу данных и таблицу, с которыми он станет работать, вызвав следующие поддерживаемые им методы:

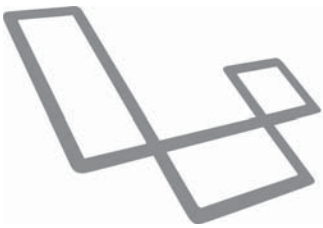
- `connection(<имя базы данных>)` — задает базу данных, с которой будет вестись работа.

Если будет вестись работа с базой данных по умолчанию, вызывать этот метод необязательно;

- `table(<имя таблицы>)` — задает обрабатываемую таблицу.

Примеры:

```
>>> use Illuminate\Support\Facades\DB;
>>> // Добавляем запись в таблицу rubrics базы данных по умолчанию
>>> DB::table('rubrics')->insert(['name' => 'Техника']);
>>> // Добавляем запись в таблицу offers базы данных mysql
>>> DB::connection('mysql')->table('offers')->insert( ... ]);
```



ГЛАВА 7

Выборка данных

7.1. Извлечение значений из полей записи

Объект модели, хранящий выбранную запись, предоставляет набор свойств, содержащих значения отдельных полей этой записи. Имена этих свойств совпадают с именами полей таблицы. Пример:

```
>>> use App\Models\Rubric;
>>> // Извлекаем первую рубрику
>>> $rubric = Rubric::first();
>>> // Получаем ее название из поля name
>>> echo $rubric->name;
Здания
>>> // Получаем ее ключ из поля id
>>> echo $rubric->id;
1
>>> // Получаем отметку правки из поля updated_at
>>> echo $rubric->updated_at;
2022-05-06 15:10:13
```

7.2. Доступ к связанным записям

7.2.1. Связь «один-со-многими»: доступ к связанным записям

В объекте записи первичной таблицы:

- обращение к свойству, чье имя совпадает с именем метода, создающего «прямую» связь, — возвращает коллекцию связанных записей вторичной таблицы, представленных в виде объектов соответствующей модели. Эту коллекцию можно перебрать в цикле, а из отдельных объектов записей извлечь значения их полей, например:

```
>>> // Посмотрим все объявления из рубрики "Дома"
>>> $rubric = Rubric::firstOrNew(['name' => 'Дома']);
>>> // В модели Rubric "прямую" связь создает метод bbs(), поэтому
>>> // для получения коллекции связанных объявлений обращаемся
>>> // к свойству bbs
```

```
>>> foreach ($rubric->bbs as $bb) {
...     echo $bb->title . ': ' . $bb->price . '   ';
... }
Дом: 5000000.0   Дом: 200000.0
```

- вызов метода, создающего «прямую» связь, — возвращает объект «прямой» связи, имеющий функциональность построителя запросов, который манипулирует лишь связанными записями. С его помощью можно сформировать запрос к связанным записям — например, отсортировать их. Пример:

```
>>> foreach ($rubric->bbs()->orderBy('price')->get() as $bb) {
...     echo $bb->title . ', ' . $bb->content . ': ' . $bb->price .
...          "\r\n";
... }
Дом, Маленький, фанерный: 200000
Дом, Большой, трехэтажный: 5000000
```

В объекте вторичной записи обращение к свойству, чье имя совпадает с именем метода, создающего «обратную» связь, — возвращает объект связанной записи первичной таблицы:

```
>>> use App\Models\Bb;
>>> // Выясним, в какой рубрике находится объявление о продаже
>>> // "Запорожца"
>>> $bb = Bb::firstOrNew(['title' => 'Запорожец']);
>>> // В модели Bb "обратную" связь создает метод rubric(), поэтому
>>> // для получения связанной рубрики обращаемся к свойству rubric
>>> echo $bb->rubric->name;
Легковой
>>> // Мы получили имя рубрики второго уровня. Теперь выясним, в какую
>>> // рубрику первого уровня она вложена.
>>> echo $bb->rubric->parent->name;
Транспорт
```

Сам метод, создающий «обратную» связь, возвращает объект «обратной» связи, также имеющий функциональность построителя запросов, и в нашем случае бесполезный.

7.2.2. Связь «один-с-одним из многих»: доступ к связанным записям

Объект первичной записи поддерживает свойство, одноименное с методом, который создает нужную связь. Это свойство хранит объект связанной записи или значение `null`, если связанных записей (по крайней мере удовлетворяющих заданным в связи условиям) нет. Пример:

```
>>> $rubric = Rubric::firstOrNew(['name' => 'Грузовой']);
>>> echo $rubric->latestBb->title, ': ', $rubric->latestBb->content;
ГАЗ: Совсем новый
>>> echo $rubric->latestMinPriceBbOnLastMonth->title, ': ',
...      $rubric->latestMinPriceBbOnLastMonth->content;
ЗИЛ: Старый, ржавый
```

7.2.3. Связь «один-с-одним»: доступ к связанным записям

Оба объекта связанных записей — первичной и вторичной таблиц — поддерживают свойства, одноименные создающим связи методам. Они хранят объекты связанных записей или `null`, если таковых нет. Пример:

```
>>> use App\Models\User;
>>> $user = User::firstOrNew(['name' => 'editor']);
>>> echo $user->account->first_name . ' ' . $user->account->last_name;
Петр Петров
>>> use App\Models\Account;
>>> $account = Account::firstOrNew(['last_name' => 'Иванов']);
>>> echo $account->user->name . ' (. $account->user->email . ')';
admin (admin@bboard.ru)
```

7.2.4. Связь «многие-со-многими»: доступ к связанным записям

Здесь используются методы, создающие связи, и одноименные им свойства, описанные в *разд. 7.2.1*. Примеры:

```
>>> use App\Models\Machine;
>>> $machine = Machine::first();
>>> echo $machine->name;
Самосвал
>>> // Из каких деталей состоит самосвал?
>>> foreach ($machine->spares as $spare) { echo $spare->name . '   '; }
Винт   Гайка   Заклепка
>>> // То же самое, только в обратном алфавитном порядке
>>> foreach ($machine->spares()->orderBy('name', 'desc')->get()
...          as $spare) {
...     echo $spare->name . '   ';
... }
Заклепка   Гайка   Винт
>>> // В состав каких машин входят гайки?
>>> use App\Models\Spare;
>>> $spare = Spare::firstOrNew(['name' => 'Гайка']);
>>> foreach ($spare->machines as $machine)
...     { echo $machine->name . '   '; }
Самосвал
```

Объект связанной записи поддерживает свойство, хранящее объект с записью связующей таблицы. По умолчанию это свойство имеет имя `pivot`, однако его можно изменить вызовом метода `as()` у объекта связи (см. *разд. 5.4.4*). Объект связанной записи либо формируется программно, либо создается на основе связующей модели (см. *разд. 5.4.4.1*).

Не забываем, что объект связанной записи хранит только поля отметок создания, правки записи связующей таблицы (если таковые имеются) и поля, приведенные в вызове метода `withPivot()` у объекта связи (см. *разд. 5.4.4*). Пример:

```
>>> // Получаем список деталей, из которых состоит самосвал,
>>> // вместе с их количеством
>>> foreach ($machine->spares as $spare) {
...     echo $spare->name . ' - ' . $spare->pivot->cnt . ' шт.   ';
... }
Винт - 9 шт.   Гайка - 9 шт.   Заклепка - 20 шт.
```

7.3. Выборка записей: базовые средства

Выборка записей осуществляется посредством вызова методов построителя запросов, описываемых далее. Получить объект построителя запросов проще всего через модель, вызвав у нее нужный метод.

7.3.1. Выборка всех записей

Выборку всех записей из таблицы выполняет метод `all()`:

```
all([<массив с именами извлекаемых полей>=['*']])
```

В единственном параметре можно передать *массив с именами полей*, которые требуется извлечь из таблицы. Если нужно извлечь все поля, следует передать массив с единственным элементом — строкой `'*'` — или вообще не указывать параметр.

Метод возвращает коллекцию извлеченных записей, являющуюся объектом класса `Collection` (будет подробно описан в *разд. 15.1*). Каждая запись в этой коллекции представляется объектом соответствующей модели. Пример:

```
>>> foreach (Rubric::all(['name']) as $rubric) {
...     echo $rubric->name . ' ';
... }
Гаражи Грузовой Дачи Дома Здания Легковой Служебные Техника Транспорт
```

По опыту автора, для той же цели можно использовать и метод `get()`, имеющий тот же формат вызова.

7.3.2. Извлечение первой записи

Извлечь первую запись таблицы можно с помощью следующих методов:

- `first()`. Формат его вызова такой же, как и у метода `all()`. Возвращается объект записи или `null`, если таблица пуста. Пример:

  ```
  >>> $bb = Bb::first(['title', 'price']);
  >>> echo $bb->title . ': ' . $bb->price;
  Гараж: 300100.0
  ```

  Метод использует для выборки первой записи средства СУБД, посылая ей соответствующую SQL-команду. Поэтому его можно использовать даже с большими таблицами без риска переполнить оперативную память ненужными записями;

- `firstOrFail()` — то же самое, что и `first()`, только в случае отсутствия записей в таблице возбуждается исключение `Illuminate\Database\Eloquent\ModelNotFoundException`;

- `sole()` — то же самое, что и `firstOrFail()`, только в случае, если в таблице присутствует больше одной записи, возбуждается исключение `Illuminate\Database\MultipleRecordsFoundException`;
- `firstOr()` — то же самое, что и `first()`, только в случае отсутствия записей в таблице возвращается результат, возвращаемый заданной *анонимной функцией*:

```
firstOr([<массив с именами извлекаемых полей>=['*'], ]
        <анонимная функция>)
```

Анонимная функция не должна принимать параметров. Пример:

```
$rubric = Rubric::firstOr(function () {
    return new Rubric(['name' => 'Здания']);
});
```

7.3.3. Поиск записей

Для поиска и извлечения записей, удовлетворяющих заданным условиям, применяются следующие методы:

- `find()` — ищет запись или записи по заданному *ключу* (*ключам*). Форматы вызова:

```
find(<ключ записи>[, <массив с именами извлекаемых полей>=['*']])
find(<массив с ключами записей>[,
     <массив с именами извлекаемых полей>=['*']])
```

При вызове в первом формате метод возвращает объект найденной записи или `null`, если записи с заданным *ключом* нет. При вызове во втором формате возвращается коллекция найденных записей (если ни одна запись не была найдена, возвращается «пустая» коллекция). Пример:

```
>>> // Какая категория имеет ключ 1?
>>> echo Rubric::find(1)->name;
Здания
>>> // Какие категории имеют ключи 3 и 5?
>>> foreach (Rubric::find([3, 5], ['name']) as $rubric) {
...     echo $rubric->name . ' ';
... }
Гаражи Легковой
```

- `findMany()` — полностью аналогичен методу `find()`, вызванному во втором формате;
- `findOrFail()` — полностью аналогичен методу `find()`, только в случае, если ни одна запись не была найдена, возбуждается исключение `ModelNotFoundException`;
- `findOrNew()` — полностью аналогичен методу `find()`, только в случае, если запись (записи) не была найдена, возвращается объект новой «пустой» записи;
- `firstWhere()` — ищет первую запись, удовлетворяющую заданному в параметрах условию (условиям), для чего формирует SQL-запрос с командой `WHERE` формата:

```
WHERE <поле> <оператор сравнения> <сравниваемое значение>
```

Возвращает объект найденной записи или `null`, если записей, удовлетворяющих заданным условиям, нет. Поддерживаются три формата вызова:

```
firstWhere(<имя поля>[, <оператор сравнения>=null],
          <сравниваемое значение>[, <логический оператор>='and'])
firstWhere(<массив с условиями поиска записи>[, null, null,
          <логический оператор>='and'])
firstWhere(<анонимная функция>[, null, null,
          <логический оператор>='and'])
```

В первом формате вызова можно указать всего одно условие поиска, задав *имя поля* таблицы, *сравниваемое значение* и *оператор сравнения* SQL (если не указан, применяется оператор равенства =):

```
// WHERE `title` = "ЗИЛ"
$bb = Bb::firstWhere('title', 'ЗИЛ');

// WHERE `price` < 50000
$bb = Bb::firstWhere('price', '<', 50000);
```

Вызов метода `firstWhere()` можно «прицепить» к вызову метода `where()` — тогда задаваемые ими обоими условия будут объединены с применением заданного в вызове метода `firstWhere()` *логического оператора* SQL (по умолчанию: `AND`):

```
// WHERE `rubric_id` = 6 AND `price` >= 10000000
$bb = Bb::where('rubric_id', 6)->firstWhere('price', '>=', 10000000);

// WHERE `rubric_id` = 6 OR `price` >= 10000000
$bb = Bb::where('rubric_id', 6)
       ->firstWhere('price', '>=', 10000000, 'or');
```

Если же сцепить два вызова метода `firstWhere()`, повторный вызов отменит условия, заданные предыдущим вызовом.

Во втором формате вызова метода `firstWhere()` задается *массив*, содержащий условия для поиска записи. Каждый элемент такого массива должен представлять собой массив со следующими элементами: имя поля таблицы, оператор сравнения SQL (необязателен), сравниваемое значение и логический оператор SQL (необязателен). Пример:

```
// WHERE `rubric_id` = 6 OR `price` >= 10000000
$bb = Bb::firstWhere([['rubric_id', $rubric->id],
                      ['price', '>=', 10000000, 'or']]);
```

В третьем формате вызова указывается *анонимная функция*, которая должна принимать в качестве параметра новый объект построителя запросов и, вызывая у него метод `where()` или аналогичные ему, задавать условия фильтрации. Заданные в *анонимной функции* условия при переводе в SQL-код будут заключены в круглые скобки. Результат *анонимная функция* возвращать не должна. Пример:

```
// WHERE `price` <= 200000 AND (`rubric_id` = 5 OR `rubric_id` = 9)
$bb = BB::where('price', '<=', 200000)
       ->firstWhere(function ($query) {
                       $query->where('rubric_id', 5)
                             ->where('rubric_id', '=', 9, 'or');
                   }
         );
```

7.3.4. Фильтрация записей

Описанные здесь методы задают условия фильтрации записей, которые используются в составе сложных запросов. В качестве результата все эти методы возвращают текущий объект построителя запросов, что позволяет выстраивать их вызовы в цепочку:

- `where()` — отбирает записи, удовлетворяющие заданному в параметрах условию (условиям), для чего формирует SQL-запрос с командой `WHERE`. Форматы вызова такие же, как и у метода `firstWhere()` (см. *разд. 7.3.3*). Примеры:

```
// WHERE `price` >= 10000000
$bbs = Bb::where('price', '>=', 10000000)->get();

// WHERE `price` >= 10000000 AND `rubric_id` = 2
$bbs = Bb::where('price', '>=', 10000000)->where('rubric_id', 2)
                                         ->get();

// WHERE `price` >= 10000000 OR `rubric_id` = 2
$bbs = Bb::where('price', '>=', 10000000)
         ->where('rubric_id', '=', 2, 'or')->get();
// То же самое
$bbs = Bb::where([['price', '>=', 10000000],
                  ['rubric_id', '=', 2, 'or']])->get();

// WHERE `rubric_id` = 2 OR (`price` >= 10000 AND `price` >= 100000)
$bbs = BB::where('rubric_id', 2)
         ->where(function ($query) {
                     $query->where('price', '>=', 10000)
                           ->where('price', '<=', 100000);
                 }, null, null, 'or')
          ->get();
```

- `orWhere()` — то же самое, что и `where()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`. Поддерживает три формата вызова:

```
orWhere(<имя поля>[, <оператор сравнения>=null],
                     <сравниваемое значение>)
orWhere(<массив с условиями фильтрации>)
orWhere(<анонимная функция>)
```

  Пример:

```
// WHERE `price` >= 10000000 OR `rubric_id` = 2
$bbs = Bb::where('price', '>=', 10000000)->orWhere('rubric_id', 2)
         ->get();
```

- `whereNot()` — отбирает записи, наоборот, *не* удовлетворяющие заданному в параметрах условию (условиям). Форматы вызова такие же, как и у метода `where()`. Пример:

```
// WHERE NOT `price` >= 10000000
$bbs = Bb::whereNot('price', '>=', 10000000)->get();
```

- `orWhereNot()` — то же самое, что и `whereNot()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`. Форматы вызова такие же, как и у метода `orWhere()`;
- `whereColumn()` — аналогичен методу `where()`, только сравнивает значение одного поля со значением другого поля. Форматы вызова:

```
whereColumn(<имя поля 1>[, <оператор сравнения>=null], <имя поля 2>[,
            <логический оператор>='and'])
whereColumn(<массив с условиями фильтрации>[, null, null,
            <логический оператор>='and'])
```

  Пример:

```
// Какие объявления никогда не правились?
// WHERE `created_at` = `updated_at`
$bbs = Bb::whereColumn('created_at', 'updated_at')->get();
```

  Во втором формате указывается *массив*, каждый элемент которого также должен представлять собой массив с четырьмя элементами: именем первого сравниваемого поля, оператором сравнения SQL (необязателен), именем второго сравниваемого поля и логическим оператором SQL (необязателен);

- `orWhereColumn()` — то же самое, что и `whereColumn()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`. Поддерживает два формата вызова:

```
orWhereColumn(<имя поля 1>[, <оператор сравнения>=null], <имя поля 2>)
orWhereColumn(<массив с условиями фильтрации>)
```

- `whereDate()` — разновидность метода `where()` для фильтрации по значениям дат:

```
whereDate(<имя поля>[, <оператор сравнения>=null],
          <сравниваемое значение>[, <логический оператор>='and'])
```

  Сравниваемое значение даты может быть указано в виде строки формата `<год>-<номер месяца>-<число>`, объекта класса `DateTime`, встроенного в PHP, или `Carbon` (в последних случаях время игнорируется). Примеры:

```
// WHERE date(`created_at`) = "2022-05-11"
$bbs = Bb::whereDate('created_at', '2022-05-11')->get();

use Illuminate\Support\Carbon;
$bbs = Bb::whereDate('updated_at', Carbon::create(2022, 05, 11))
         ->get();
```

- `orWhereDate()` — то же самое, что и `whereDate()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`:

```
orWhereDate(<имя поля>[, <оператор сравнения>=null],
                         <сравниваемое значение>)
```

- `whereDay()` — разновидность метода `where()` для фильтрации по числам (номерам дней). Формат вызова тот же, что и у метода `whereDate()`. Пример:

```
// Какие объявления были исправлены после 14 числа?
// WHERE day(`updated_at`) > 14
$bbs = Bb::whereDay('updated_at', '>', 14)->get();
```

- `orWhereDay()` — то же самое, что и `whereDay()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereDate()`;
- `whereMonth()` — разновидность метода `where()` для фильтрации по номеру месяца. Формат вызова тот же, что и у метода `whereDate()`;
- `orWhereMonth()` — то же самое, что и `whereMonth()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereDate()`;
- `whereYear()` — разновидность метода `where()` для фильтрации по году. Формат вызова тот же, что и у метода `whereDate()`;
- `orWhereYear()` — то же самое, что и `whereYear()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereDate()`;
- `whereTime()` — разновидность метода `where()` для фильтрации по значениям времени. Формат вызова тот же, что и у метода `whereDate()`. Сравниваемое значение времени может быть указано в виде строки формата *<часы>*:*<минуты>*:*<секунды>*, объекта класса `DateTime`, встроенного в PHP, или `Carbon` (в последних случаях дата игнорируется). Примеры:

```
// Какие объявления были исправлены до полудня?
// WHERE time(`updated_at`) < "12:00:00"
$bbs = Bb::whereTime('updated_at', '<',
                     new DateTime('2022-06-18 12:00:00'))->get();
```

- `orWhereTime()` — то же самое, что и `whereTime()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereDate()`;
- `whereBetween()` — отбирает записи, у которых поле с указанным *именем* хранит значения, укладывающиеся в заданный диапазон:

```
whereBetween(<имя поля>, <массив с граничными значениями диапазона>[,
     <логический оператор>='and'[, <инвертировать результат?>=false]])
```

Массив с граничными значениями должен содержать два элемента: начальное и конечное значения диапазона. Назначение параметра *логический оператор* то же, что и у метода `firstWhere()`. Если параметру *инвертировать результат* дать значение `true`, метод будет отбирать записи, у которых значение заданного поля, наоборот, *не* входит в указанный диапазон. Примеры:

```
// WHERE `price` BETWEEN 10000 AND 100000
$bbs = Bb::whereBetween('price', [10000, 100000])->get();

// WHERE `price` NOT BETWEEN 10000 AND 100000
$bbs = Bb::whereBetween('price', [10000, 100000], 'and', true)->get();
```

- `whereNotBetween()` — отбирает записи, у которых поле с указанным *именем* хранит значения, *не* укладывающиеся в заданный диапазон. В остальном аналогичен методу `whereBetween()`. Формат вызова:

```
whereNotBetween(<имя поля>,
                <массив с граничными значениями диапазона>[,
                <логический оператор>='and'])
```

- `orWhereBetween()` — то же самое, что и `whereBetween()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`:

```
orWhereBetween(<имя поля>, <массив с граничными значениями диапазона>)
```

- `orWhereNotBetween()` — то же самое, что и `whereNotBetween()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereBetween()`;
- `whereBetweenColumns()` — отбирает записи, у которых поле с указанным *именем* хранит значения, укладывающиеся в диапазон, который задан значениями из полей, перечисленных в *массиве*:

```
whereBetweenColumns(<имя поля>, <массив с именами полей>[,
     <логический оператор>='and'[, <инвертировать результат?>=false]])
```

Массив должен содержать два имени поля: первое задаст начальную границу диапазона, второе — конечную. Назначение параметра *логический оператор* то же, что и у метода `firstWhere()`, назначение параметра *инвертировать результат* — как и у метода `whereBetween()`. Пример:

```
// WHERE `offered_price` BETWEEN `min_price` AND `max_price`
$bbs = Bb::whereBetweenColumns('offered_price',
                               ['min_price', 'max_price'])
          ->get();
```

- `whereNotBetweenColumns()` — отбирает записи, у которых поле с указанным *именем* хранит значения, *не* укладывающиеся в диапазон, который задан значениями из полей, присутствующих в *массиве*:

```
whereNotBetweenColumns(<имя поля>, <массив с именами полей>[,
                       <логический оператор>='and'])
```

- `orWhereBetweenColumns()` — то же самое, что и `whereBetweenColumns()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`:

```
orWhereBetweenColumns(<имя поля>, <массив с именами полей>)
```

- `orWhereNotBetweenColumns()` — то же самое, что и `whereNotBetweenColumns()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereBetweenColumns()`;
- `whereIn()` — отбирает записи, у которых поле с указанным *именем* хранит одно из значений, присутствующих в заданном *массиве*. Формат вызова:

```
whereIn(<имя поля>, <массив со значениями>[,
     <логический оператор>='and'[, <инвертировать результат?>=false]])
```

Назначение параметра *логический оператор* то же, что и у метода `firstWhere()`. Если параметру *инвертировать результат* присвоить значение `true`, метод будет отбирать записи, у которых заданное поле хранит значения, наоборот, *не* присутствующие в *массиве*. Примеры:

```
// WHERE `rubric_id` IN (2, 3, 5)
$bbs = Bb::whereIn('rubric_id', [2, 3, 5])->get();

// WHERE `title` IN ("ЗИЛ", "Дом")
$bbs = Bb::whereIn('title', ['ЗИЛ', 'Дом'])->get();
```

- `whereIntegerInRaw()` — то же самое, что и `whereIn()`, но позволяет выполнять фильтрацию только по целочисленным значениям и выполняется несколько быстрее, в особенности при указании больших массивов со значениями:

```
$bbs = Bb::whereIntegerInRaw('rubric_id', [2, 3, 5])->get();
```

- `whereNotIn()` — фильтрует записи, у которых поле с указанным *именем* хранит одно из значений, *не* присутствующих в заданном *массиве*. В остальном аналогичен методу `whereIn()`. Формат вызова:

```
whereNotIn(<имя поля>, <массив со значениями>[,
                       <логический оператор>='and'])
```

- `whereIntegerNotInRaw()` — то же самое, что и `whereNotIn()`, но позволяет выполнять фильтрацию только по целочисленным значениям. Выполняется несколько быстрее, чем `whereNotIn()`;
- `orWhereIn(<имя поля>, <массив со значениями>)` — то же самое, что и `whereIn()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`;
- `orWhereIntegerInRaw()` — то же самое, что и `orWhereIn()`, но позволяет выполнять фильтрацию только по целочисленным значениям. Выполняется несколько быстрее, чем `onWhereIn()`;
- `orWhereNotIn()` — то же самое, что и `whereNotIn()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereIn()`;
- `orWhereIntegerNotInRaw()` — то же самое, что и `orWhereNotIn()`, но позволяет выполнять фильтрацию только по целочисленным значениям. Выполняется несколько быстрее, чем `onWhereNotIn()`;
- `whereNull()` — отбирает записи, у которых поле с заданным *именем* (или все поля с именами, присутствующими в *массиве*) хранит значение `null`:

```
whereNull(<имя поля>|<массив с именами полей>[,
     <логический оператор>='and'[, <инвертировать результат?>=false]])
```

Назначение параметра *логический оператор* то же, что и у метода `firstWhere()`. Если параметру *инвертировать результат* присвоить значение `true`, метод будет отфильтровывать записи, у которых заданное поле (или все заданные поля), наоборот, хранит значение, *отличное от* `null`. Пример:

```
// Выбираем все рубрики первого уровня
// WHERE `parent_id` IS NULL
$rubrics = Rubric::whereNull('parent_id')->get();
```

- `whereNotNull()` — отбирает записи, у которых поле с заданным *именем* (или все поля с именами, указанными в *массиве*) хранит значение, *отличное от* `null`. В остальном аналогичен методу `whereNull()`. Формат вызова:

```
whereNotNull(<имя поля>|<массив с именами полей>[,
             <логический оператор>='and'])
```

Пример:

```
// Выбираем все рубрики второго уровня
// WHERE `parent_id` IS NOT NULL
$rubrics = Rubric::whereNotNull('parent_id')->get();
```

- `orWhereNull(<имя поля>)` — то же самое, что и `whereNull()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`;
- `orWhereNotNull(<имя поля>)` — то же самое, что и `whereNotNull()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`.

7.3.4.1. Фильтрация записей по значениям полей типа *JSON*

При фильтрации по значениям свойств объектов, хранящихся в полях типа `JSON`, для доступа к этим свойствам следует применять принятую в PHP «стрелочную» запись:

```
$bbs = Bb::where('state->reason', 'Устарело')->get();
```

> ***Для фильтрации по полям типа JSON в базах данных SQLite...***
> ...необходимо использовать расширение JSON1 этой СУБД.

Специально для выполнения фильтрации по значениям свойств объектов из полей типа `JSON` построитель запросов предоставляет следующие методы:

- `whereJsonContainsKey()` — отбирает записи, хранящие свойство с указанным путем:

```
whereJsonContainsKey(<путь к свойству>[, <логический оператор>='and'[,
                     <инвертировать результат?>=false]])
```

Назначение параметра *логический оператор* то же, что и у метода `firstWhere()` (см. *разд. 7.3.3*). Если параметру *инвертировать результат* дать значение `true`, метод будет отбирать записи, наоборот, *не* содержащие свойство с указанным *путем*. Пример:

```
$bbs = Bb::whereJsonContainsKey('state->addendum')->get();
```

- `whereJsonDoesntContainKey()` — отбирает записи, наоборот, *не* хранящие свойство с указанным путем:

```
whereJsonDoesntContainKey(<путь к свойству>[,
                          <логический оператор>='and'])
```

- `orWhereJsonContainsKey(<путь к свойству>)` — то же самое, что и `whereJsonContainsKey()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`;
- `orWhereJsonDoesntContainKey()` — то же самое, что и `whereJsonDoesntContainKey()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereJsonContainsKey()`;
- `whereJsonContains()` — отбирает записи, у которых массив, хранящийся в свойстве с указанным *путем*, содержит указанное *значение*:

```
whereJsonContains(<путь к свойству>, <сравниваемое значение>[,
    <логический оператор>='and'[, <инвертировать результат?>=false]])
```

Назначение параметра *логический оператор* то же, что и у метода `firstWhere()` (см. *разд. 7.3.3*). Если параметру *инвертировать результат* дать значение `true`, метод будет отбирать записи, у которых массив, хранящийся в заданном свойстве, наоборот, *не* содержит указанное *значение*. Пример:

```
$bbs = Bb::whereJsonContains('state->options', 'срочное')->get();
```

В случае использования базы данных форматов MySQL или PostgreSQL вторым параметром можно указать массив сравниваемых значений:

```
$bbs = Bb::whereJsonContains('state->options',
                             ['срочное', 'весьма срочное'])->get();
```

- `whereJsonDoesntContain()` — отбирает записи, у которых массив, хранящийся в свойстве с указанным *путем*, наоборот, *не* содержит заданное *значение*. В остальном аналогичен методу `whereJsonContains()`. Формат вызова:

```
whereJsonDoesntContain(<путь к свойству>, <сравниваемое значение>[,
                       <логический оператор>='and'])
```

- `orWhereJsonContains()` — то же самое, что и `whereJsonContains()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`:

```
orWhereJsonContains(<путь к свойству>, <сравниваемое значение>)
```

- `orWhereJsonDoesntContain()` — то же самое, что и `whereJsonDoesntContain()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`. Формат вызова тот же, что и у метода `orWhereJsonContains()`;

- `whereJsonLength()` — отбирает записи, у которых результат сравнения длины массива, хранящегося в свойстве с указанным *путем*, с заданным *значением* даст положительный результат:

```
whereJsonLength(<путь к свойству>[, <оператор сравнения>=null],
        <сравниваемое значение>[, <логический оператор>='and'])
```

Назначение параметров *оператор сравнения* и *логический оператор* то же, что и у метода `firstWhere()`. Пример:

```
$bbs = Bb::whereJsonLength('state->options', '>', 1)->get();
```

- `orWhereJsonLength()` — то же самое, что и `whereJsonLength()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`:

```
orWhereJsonLength(<путь к свойству>[, <оператор сравнения>=null],
                                    <сравниваемое значение>)
```

7.3.4.2. Фильтрация по полнотекстовому индексу

Для фильтрации записей по значениям полей, на основе которых были созданы полнотекстовые индексы (см. *разд. 4.1.3.4*), служат два описываемых далее метода. Они возвращают текущий объект построителя запросов:

- `whereFullText()` — отбирает записи, у которых заданное *поле* (или *поля*) содержат указанное *значение*:

```
whereFullText(<имя поля>|<массив с именами полей>,
              <искомое значение>[, <массив с параметрами>=[][,
              <логический оператор>='and']])
```

Первым параметром можно указать как *имя поля*, так и *массив с именами полей*, если требуется выполнять фильтрацию по значениям сразу нескольких полей. *Искомое значение* должно иметь строковый тип.

Третьим параметром можно передать ассоциативный *массив* с дополнительными параметрами выполняемой фильтрации. Ключи элементов этого массива должны соответствовать параметрам, а значения элементов зададут значения параметров. В настоящее время обрабатываются параметры `mode` (поддерживается MySQL и PostgreSQL), `expanded` (MySQL) и `language` (PostgreSQL).

Назначение параметра *логический оператор* то же, что и у метода `firstWhere()` (см. *разд. 7.3.3*).

Примеры:

```
$bbs = Bb::whereFullText('content', 'новый');

$bbs = Bb::whereFullText(['title', 'content'], 'дом',
                         ['mode' =>' natural']);
```

- `orWhereFullText()` — то же самое, что и `whereFullText()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`:

```
orWhereFullText(<имя поля>|<массив с именами полей>,
                <искомое значение>[, <массив с параметрами>=[]])
```

7.3.5. Сортировка записей

Для сортировки записей построитель запросов предоставляет методы, описанные далее. Все они возвращают текущий объект построителя запросов:

- `orderBy()` — сортирует записи по значению поля с указанным *именем* в заданном *направлении*:

```
orderBy(<имя поля>[, <направление>='asc'])
```

Значение `'asc'` параметра *направление* задает сортировку по возрастанию значения поля, значение `'desc'` — по убыванию. Пример:

```
// ORDER BY `rubric_id`
$bbs = Bb::orderBy('rubric_id')->get();
```

Чтобы отсортировать записи по нескольким полям, следует записать «цепочку» вызовов метода `orderBy()`:

```
// ORDER BY `rubric_id`, `price` DESC
$bbs = Bb::orderBy('rubric_id')->orderBy('price', 'desc')->get();
```

- `orderByDesc(<имя поля>)` — аналогичен `orderBy()`, но сортирует только по убыванию значения поля с заданным *именем*:

```
$bbs = Bb::orderBy('rubric_id')->orderByDesc('price')->get();
```

- `reorder()` — то же самое, что и `orderBy()`, только предварительно удаляет все ранее заданные сортировки:

  ```
  reorder([<имя поля>[, <направление>='asc']])
  ```

  Если *имя поля* не задано, записи останутся несортированными. Пример:

  ```
  // В результате объявления будут отсортированы только по убыванию цены
  $bbs = Bb::orderBy('rubric_id')->reorder('price', 'desc')->get();
  ```

- `latest([<имя поля>='created_at'])` — сортирует записи по убыванию значения поля с указанным *именем*, которое должно хранить временны́е отметки;
- `oldest([<имя поля>='created_at'])` — сортирует записи по возрастанию значения поля с указанным *именем*, которое должно хранить временны́е отметки;
- `inRandomOrder([<начальное значение>=''])` — сортирует записи в псевдослучайном порядке. Можно указать строковое *начальное значение*, используемое генератором псевдослучайных чисел (по умолчанию: «пустая» строка). Пример:

  ```
  // Извлекаем произвольное объявление
  $bb = Bb::inRandomOrder()->first();
  ```

7.3.6. Выборка указанного количества записей

Выборку указанного количества записей с пропуском заданного числа записей выполняют представленные далее методы построителя запросов. Все они возвращают текущий объект построителя запросов:

- `limit(<количество выбираемых записей>)` — выбирает указанное *количество записей*:

  ```
  >>> $rubrics = Rubric::get();
  >>> foreach ($rubrics as $rubric) { echo $rubric->name . ' '; }
  Здания Дома Гаражи Транспорт Легковой Грузовой Дачи Служебные Техника
  >>> $rubrics = Rubric::limit(3)->get();
  . . .
  Здания Дома Гаражи
  ```

- `take(<количество выбираемых записей>)` — то же самое, что и `limit()`;
- `offset(<количество пропускаемых записей>)` — пропускает заданное *количество записей* перед выборкой. Применяется только вместе с методом `limit()` или `take()`. Пример:

  ```
  >>> $rubrics = Rubric::limit(3)->offset(2)->get();
  . . .
  Гаражи Транспорт Легковой
  ```

- `skip(<количество пропускаемых записей>)` — то же самое, что и `offset()`.

7.3.7. Выборка уникальных записей

Чтобы выбрать только уникальные записи, следует вызвать метод `distinct()` построителя запросов. В качестве результата он возвращает текущий объект построителя запросов. Пример:

```
>>> $bbs = Bb::get(['rubric_id']);
>>> foreach ($bbs as $bb) { echo $bb->rubric->name . ' '; }
Гаражи Дачи Дома Дома Легковой Грузовой Легковой Грузовой Грузовой
>>> $bbs = Bb::distinct()->get(['rubric_id']);
. . .
Гаражи Дачи Дома Легковой Грузовой
```

7.3.8. Задание параметров запросов на основании выполнения указанного условия

Иногда бывает необходимо задать одни параметры формируемого запроса к базе данных, если какое-либо условие выполняется, и другие — если оно не выполняется. В таком случае поможет метод `when()` построителя запросов, возвращающий его текущий объект:

```
when(<условное выражение>, <анонимная функция 1>[,
                           <анонимная функция 2>=null])
```

Если результатом *условного выражения* является `true`, выполняется *анонимная функция 1*, в противном случае — *анонимная функция 2* (если она не указана, ничего не происходит). Обе *анонимные функции* должны принимать два параметра: текущий объект построителя запросов и результат вычисления *условного выражения*. В теле этих *функций* следует задать нужные параметры запроса посредством полученного с первым параметром построителя запросов, после чего вернуть его в качестве результата.

Пример:

```
// Если в клиентском HTTP-запросе присутствует параметр reverse,
// сортируем объявления в прямом хронологическом порядке,
// в противном случае — в обратном порядке
$bbs = Bb::when($request->reverse,
                function ($query, $reverse) { return $query->oldest(); },
                function ($query, $reverse) { return $quesy->latest(); })
        ->get();
```

7.3.9. Смена типа выдаваемых значений

Привести значения каких-либо полей, присутствующих в результате запроса, к другому типу можно вызовом метода `withCasts(<параметры преобразования типов>)` построителя запросов. *Параметры преобразования* указываются в виде массива, аналогичного заносимому в свойство `casts` модели (см. *разд. 5.3.3*). Пример:

```
>>> $price = Bb::first()->price;
=> 300100.0
>>> $price = Bb::withCasts(['price' => 'int'])->first()->price;
=> 300100
```

7.3.10. Выполнение запроса и получение результата

После указания всех необходимых параметров запроса к базе данных его надо выполнить и получить результат. Это осуществляется с помощью следующих методов построителя запросов:

- `get([<массив с именами извлекаемых полей>=['*']])` — выполняет запрос и возвращает результат его исполнения — коллекцию объектов модели. Можно указать *массив с именами извлекаемых полей*, если же передать массив с единственным элементом `'*'`, будут извлечены все поля;
- `first()`, `firstOrFail()` и `firstOr()` — выполняют запрос и возвращают первую запись из полученного набора (эти методы были описаны в *разд. 7.3.2*);
- `sole()` — выполняет запрос и возвращает единственную запись из полученного набора (см. *разд. 7.3.2*).

7.3.11. Проверка наличия записей в полученном результате

Если нужно просто выяснить, существуют ли записи в выданном SQL-запросом результате, применяют следующие методы построителя запросов:

- `exists()` — возвращает `true`, если в результате, выданном запросом, есть записи, и `false` — в противном случае:

```
>>> $bbsExists = Bb::where('title', 'Дом')->exists();
=> true
>>> $bbsExists = Bb::where('title', 'Сарай')->exists();
=> false
```

- `doesntExist()` — возвращает `true`, если в результате, выданном запросом, наоборот, *нет* записей, и `false` — в противном случае;
- `existsOr(<анонимная функция>)` — возвращает `true`, если в результате, выданном запросом, есть записи, и результат исполнения не принимающей параметров *анонимной функции* — в противном случае;
- `doesntExistOr(<анонимная функция>)` — возвращает `true`, если в результате, выданном запросом, наоборот, *нет* записей, и результат исполнения не принимающей параметров *анонимной функции* — в противном случае:

```
$r = Rubric::where('name', 'Инвентарь')
           ->doesntExistOr(function () {
                          return new Rubric(['name' => 'Инвентарь']);
                          }
             );
```

7.3.12. Объединение результатов от разных запросов

Чтобы объединить результаты, выдаваемые двумя разными запросами, в единую коллекцию, надо вызвать один из двух следующих методов построителя запросов:

- `union()` — объединяет результаты, выдаваемые текущим и указанным в первом параметре построителями запросов:

```
union(<построитель запросов>[,
      <включать повторяющиеся записи?>=false])
```

Если вторым параметром передать значение `true`, результат включит все записи, выдаваемые обоими запросами, — даже повторяющиеся (по умолчанию в результат включаются только уникальные записи).

Метод возвращает текущий объект построителя запросов. Вызвав у последнего метод `get()` (или любой другой, описанный в *разд. 7.3.9* и *7.3.10*), можно получить результат объединения результатов. Пример:

```
>>> // Объединяем результаты выборки объявлений из рубрики "Грузовой"
>>> // и объявлений с заявленной ценой не более 100 тыс. руб.
>>> $rubric = Rubric::firstWhere('name', 'Грузовой');
>>> $qb = Bb::where('price', '<=', 100000);
>>> $bbs = Bb::where('rubric_id', $rubric->id)->union($qb)->get();
>>> foreach ($bbs as $bb) {
...     echo $bb->rubric->name . ': ' . $bb->title . ' (' .
...                                     $bb->price . ")\r\n";
... }
Дачи: Дача (100000)
Грузовой: ЗИЛ (1000000)
Легковой: Запорожец (10000)
Грузовой: МАЗ (4000000)
Грузовой: ГАЗ (70000000)
```

- `unionAll(<`*построитель запросов*`>)` — то же самое, что и `union()`, только всегда включает в результат все записи, в том числе и повторяющиеся.

7.4. Выборка связанных записей

Объект связи, возвращаемый методами модели, которые создают связи, имеет функциональность построителя запросов. Следовательно, для сортировки и фильтрации связанных записей можно использовать методы, описанные в *разд. 7.3*. Пример:

```
>>> $rubric = Rubric::firstWhere('name', 'Грузовой');
>>> $bbs = $rubric->bbs()->orderBy('price', 'desc')->get();
>>> foreach ($bbs as $bb) {
...     echo $bb->title . ': ' . $bb->price . '   ';
... }
ГАЗ: 70000000   МАЗ: 4000000   ЗИЛ: 1000000
>>> $bbs = $rubric->bbs()->where('price', '>', 10000000)->get();
. . .
ГАЗ: 70000000
```

Однако здесь есть одна тонкость. Для выборки связанных записей Laravel вставляет в формируемый запрос SQL-команду `WHERE` с условием формата *<поле внешнего ключа> = <ключ записи первичной таблицы>*. Последующий вызов метода `where()` добавляет следующее условие посредством логического оператора `AND`. Однако если далее идет вызов метода `orWhere()`, очередное условие будет добавлено уже посредством оператора `OR`, имеющего более низкий приоритет исполнения, чем `AND`. Вследствие чего возникнет следующая ситуация:

```
$bbs = $rubric->bbs()->where('title', 'ЗИЛ')
                     ->orWhere('price', '>', 4000000)->get();
```

```
// В результате будет сгенерирован следующий SQL-запрос:
// SELECT * FROM bbs WHERE rubric_id = <ключ рубрики> AND
//                         title = 'ЗИЛ' OR price > 4000000
// И возвращаемый запросом результат будет далек от ожидаемого
```

Чтобы исключить такую ситуацию, следует использовать третий формат вызова методов `where()` (см. *разд. 7.3.4*):

```
$bbs = $rubric->bbs()
             ->where(function ($query) {
                        return $query->where('title', 'ЗИЛ')
                                     ->orWhere('price', '>', 4000000);
                    }
              )->get();
// В результате будет сгенерирован SQL-запрос:
// SELECT * FROM bbs WHERE rubric_id = <ключ рубрики> AND
//                          (title = 'ЗИЛ' OR price > 4000000)
```

Можно отбирать только те записи первичной модели, у которых хотя бы одна связанная запись вторичной модели хранит в заданном поле указанное значение. Для этого применяются методы:

- `whereRelation()` — отбирает записи первичной таблицы, у которых сравнение значения поля с указанным *именем* хотя бы одной записи вторичной таблицы с заданным *значением* с использованием указанного *оператора сравнения* дает положительный результат. Форматы вызова:

  ```
  whereRelation(<имя связи с вторичной моделью>,
                <имя поля вторичной модели>[,
                <оператор сравнения>=null], <сравниваемое значение>)
  whereRelation(<имя связи с вторичной моделью>,
                <анонимная функция, отбирающая связанные записи>)
  ```

  Имя связи совпадает с именем метода модели, создающего «прямую» связь (например, если «прямая» связь создается методом `bbs()`, сама связь также имеет имя `bbs`).

  В первом формате вызова можно указать всего одно условие фильтрации, задав *имя поля* таблицы, *сравниваемое значение* и *оператор сравнения* SQL (если не указан, применяется оператор равенства `=`):

  ```
  // Извлекаем рубрики, имеющие хотя бы одно объявление
  // с заявленной ценой более 1 млн руб.
  $bbs = Rubric::whereRelation('bbs', 'price', '>', 1000000)->get();
  ```

  Во втором формате вызова указывается *анонимная функция*, которая должна принимать в качестве параметра новый объект построителя запросов и задавать с его помощью условия фильтрации:

  ```
  // Извлекаем рубрики, имеющие хотя бы одно объявление без указанного
  // адреса
  Rubric::whereRelation('bbs',
                        function ($query) {
                            $query->whereNull('address');
                        }
                       )->get();
  ```

Вызовы метода `whereRelation()` можно «сцеплять» друг с другом — тогда задаваемые ими условия будут объединены с применением логического оператора `AND`;

- `orWhereRelation()` — то же самое, что и `whereRelation()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`.

Также можно выполнять фильтрацию записей первичной таблицы по наличию, отсутствию записей вторичной таблицы или отбирать записи первичной таблицы, с которыми связано определенное количество записей вторичной таблицы. Для этого применяются следующие методы, поддерживаемые построителем запросов и возвращающие его текущий объект в качестве результата:

- `has()` — отбирает записи первичной таблицы, у которых сравнение количества связанных с ними записей вторичной таблицы с указанным *значением* с использованием заданного *оператора сравнения* дает положительный результат:

```
has(<имя связи с вторичной моделью>[, <оператор сравнения>='>='[,
    <количество связанных записей>=1[, <логический оператор>='and'[,
    <анонимная функция, отбирающая связанные записи>=null]]]])
```

Имя связи совпадает с именем метода модели, создающего «прямую» связь (например, если «прямая» связь создается методом `bbs()`, сама связь также имеет имя `bbs`). Можно указать и *логический оператор* SQL, посредством которого условие, задаваемое текущим вызовом метода `has()`, связывается с условием, задаваемым предыдущим вызовом этого метода.

Анонимная функция, указываемая последним параметром, может задавать условия фильтрации записей вторичной таблицы — в этом случае метод `has()` будет подсчитывать лишь записи, удовлетворяющие заданным условиям. Эта функция должна принимать с параметром объект построителя запросов и задавать с его помощью условия фильтрации. Примеры:

```
// Извлекаем рубрики второго уровня, имеющие хотя бы одно объявление
$rubrics = Rubric::whereNotNull('parent_id')->has('bbs')->get();

// Извлекаем рубрики, имеющие больше одного объявления
$rubrics = Rubric::whereNotNull('parent_id')->has('bbs', '>', 1)
                                            ->get();

// Извлекаем рубрики, имеющие больше одного объявления
// с заявленной ценой не менее 5 млн руб.
$rubrics = Rubric::whereNotNull('parent_id')
                 ->has('bbs', '>', 1, 'and',
                       function ($query) {
                           $query->where('price', '>=', 5000000);
                       }
                   )->get();
```

Если связанная вторичная модель, в свою очередь, связана с какой-либо другой (третичной) моделью связью «один-со-многими», можно выполнять фильтрацию по количеству записей третичной модели, записав имя связи в формате:

```
<имя связи с вторичной моделью>.<имя связи с третичной моделью>
```

Пример:

```
// Извлекаем рубрики первого уровня, у которых вложенные рубрики
// второго уровня имеют хотя бы одно объявление
$rubrics = Rubric::whereNull('parent_id')->has('rubrics.bbs')->get();
```

- `orHas()` — то же самое, что и `has()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`:

```
orHas(<имя связи с вторичной моделью>[, <оператор сравнения>='>='[,
      <количество связанных записей>=1]])
```

- `doesntHave()` — отбирает записи первичной таблицы, у которых нет подходящих связанных записей вторичной таблицы:

```
doesntHave(<имя связи с вторичной моделью>[,
           <логический оператор>='and'[,
           <анонимная функция, фильтрующая связанные записи>=null]])
```

Пример:

```
// Извлекаем рубрики второго уровня, у которых нет объявлений
// с заявленной ценой более 1 млн руб.
$rubrics = Rubric::whereNotNull('parent_id')
         ->doesntHave('bbs', 'and',
                 function ($query) {
                          $query->where('price', '>=', 1000000);
                 }
           )->get();
```

- `orDoesntHave(<имя связи>)` — то же самое, что и `doesntHave()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`;
- `whereHas()` — то же самое, что и `has()`, но с другим форматом вызова, более удобным в случае использования *анонимной функции*:

```
whereHas(<имя связи с вторичной моделью>[,
         <анонимная функция, отбирающая связанные записи>=null[,
         <оператор сравнения>='>='[,
         <количество связанных записей>=1]]])
```

Пример:

```
$rubrics = Rubric::whereNotNull('parent_id')
         ->whereHas('bbs', function ($query) {
                              $query->where('price', '>=', 5000000);
                          },
                    '>', 1)
         ->get();
```

- `orWhereHas()` — то же самое, что и `whereHas()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`;
- `whereDoesntHave()` — то же самое, что и `doesntHave()`, но с другим форматом вызова, более удобным в случае использования *анонимной функции*:

```
whereDoesntHave(<имя связи с вторичной моделью>[,
                <анонимная функция, отбирающая связанные записи>=null])
```

Пример:

```
$rubrics = Rubric::whereNotNull('parent_id')
         ->whereDoesntHave('bbs',
                function ($query) {
                         $query->where('price', '>=', 1000000);
                }
           )->get();
```

- `orWhereDoesntHave()` — то же самое, что и `whereDoesntHave()`, только задаваемое им условие объединяется с предыдущим с использованием логического оператора `OR`.

7.5. Выборка записей: расширенные средства

Рассматриваемые здесь расширенные средства позволяют создавать более сложные запросы к базам данных, включающие вызов встроенных функций СУБД.

7.5.1. Указание выбираемых полей

Указать поля, значения которых будут выбираться из таблицы, можно в вызове методов: `all()`, `get()`, `first()`, `find()` и др., описанных в *разд. 7.3*. Но методы построителя запросов, приведенные далее, предоставляют дополнительные возможности:

- `select()` — задает поля, выбираемые из таблицы. Поддерживаются два формата вызова:

  ```
  select(<имя поля 1>, <имя поля 2>, ... <имя поля n>)
  select(<массив с именами полей>)
  ```

  Можно указать как непосредственно имя поля (тогда его значение можно получить из одноименного свойства объекта модели), так и строку формата `<имя поля> as <псевдоним>` — после чего значение этого поля будет храниться в свойстве, чье имя совпадает с заданным *псевдонимом*. Пример:

  ```
  >>> $bbs = Bb::select('title', 'content')->get();
  >>> foreach ($bbs as $bb) {
  ...     echo $bb->title . ': ' . $bb->content . "\r\n";
  ... }
  Гараж: На две машины
  Дача: Новая
  . . .
  ГАЗ: Совсем новый
  >>> $bbs = Bb::select(['title', 'content as description'])->get();
  >>> foreach ($bbs as $bb) {
  ...     echo $bb->title . ': ' . $bb->description . "\r\n";
  ... }
  Гараж: На две машины
  . . .
  ```

  Следующий вызов метода `select()` отменяет все поля, заданные его предыдущим вызовом;

- `addSelect()` — то же самое, что и `select()`, только добавляет указанные в его вызове поля к заданным предыдущими вызовами методам `select()` и `addSelect()`:

```
$bbs = Bb::select('title')->addSelect(['content as description'])
                          ->get();
```

7.5.2. Вставка фрагментов SQL-кода в запрос

Все описанные ранее методы построителя запросов, задающие условия фильтрации и сортировки, вместо имени поля позволяют указать объект, представляющий фрагмент SQL-кода (например, вызов встроенной функции СУБД). Получить такой объект можно вызовом метода `raw(<строка с SQL-кодом>)` у фасада `DB`. Примеры:

```
>>> // Выводим все объявления о продажах товаров, чьи названия длиннее
>>> // 5 символов. Для вычисления длины строки используем функцию
>>> // length() SQLite.
>>> use Illuminate\Support\Facades\DB;
>>> $bbs = Bb::where(DB::raw('length(title)'), '>', 5)->get();
>>> foreach ($bbs as $bb) { echo $bb->title . '   '; }
Жигули   Запорожец

>>> // Сортируем рубрики по убыванию длины их названий
>>> $rubrics = Rubric::orderBy(DB::raw('length(name)'), 'desc')->get();
>>> foreach ($rubrics as $rubric) { echo $rubric->name . '   '; }
Транспорт   Служебные   Легковой   Грузовой   Техника   Здания   ↳
Гаражи   Дома   Дачи

>>> // Функция printf() SQLite форматирует значения заданных полей
>>> // согласно строке формата, указанной в первом параметре
>>> $bbs = Bb::select(
...          DB::raw('printf("%!-12s %10.0f", title, price) as output'))
...      ->get();
>>> foreach ($bbs as $bb) { echo $bb->output . "\r\n"; }
Гараж            300100
Дача             100000
. . .
ГАЗ            70000000
```

7.5.3. Связывание таблиц

Для извлечения значений полей из связанных записей удобно использовать средства, описанные в *разд. 7.2*. Однако при этом для извлечения связанных записей Laravel выполняет дополнительный запрос к базе данных, что снижает производительность. В таких случаях целесообразнее явно задать связь между таблицами при формировании запроса.

> ***ПРИ СВЯЗЫВАНИИ ТАБЛИЦ СЛЕДУЕТ ЯВНО УКАЗАТЬ ВСЕ ИЗВЛЕКАЕМЫЕ ПОЛЯ!***
>
> Задать поля, извлекаемые из таблиц, как текущей, так и связанных, можно в вызовах методов `select()` и `addSelect()`.

Для установления связей между таблицами построитель запросов предлагает следующие методы:

- `join()` — устанавливает связь между текущей таблицей и таблицей с указанным *именем* на основе совпадения значений полей с заданными *именами*, принадлежащих разным таблицам.

  Поддерживаются два формата вызова:

```
join(<имя связываемой таблицы>, <имя сравниваемого поля 1>,
     <оператор сравнения>, <имя сравниваемого поля 2>[,
     <тип связи>='inner'])
join(<имя связываемой таблицы>, <анонимная функция>[, null, null,
     <тип связи>='inner'])
```

  В первом формате вызова все параметры устанавливаемой связи указываются непосредственно. *Тип связи* может быть `'inner'` (внутренняя, создается по умолчанию), `'left'` (левая), `'right'` (правая) и `'cross'` (перекрестная). Пример:

```
>>> $bbs = Bb::select('bbs.*', 'rubrics.name as rubric_name')
...      ->join('rubrics', 'bbs.rubric_id', '=', 'rubrics.id')->get();
>>> foreach ($bbs as $bb) {
...     echo $bb->title . ': ' . $bb->price . ' (' .
...                              $bb->rubric_name . ")\r\n";
... }
Гараж: 300100 (Гаражи)
Дача: 100000 (Дачи)
Дом: 5000000 (Дома)
Дом: 200000 (Дома)
Жигули: 1000000 (Легковой)
. . .
```

  Вызовы метода `join()` можно сцеплять друг с другом, чтобы установить связи сразу с несколькими таблицами.

  Вместо *имени связываемой таблицы* можно указать строку формата *<имя таблицы>* as *<псевдоним>* — тогда связываемая таблица будет доступна под заданным *псевдонимом*:

```
>>> $bbs = Bb::select('bbs.*', 'rubrics.name as rubric_name',
...                   'superrubrics.name as superrubric_name')
...      ->join('rubrics','bbs.rubric_id', '=', 'rubrics.id')
...      ->join('rubrics as superrubrics', 'rubrics.parent_id', '=',
...                                        'superrubrics.id')
...      ->get();
>>> foreach ($bbs as $bb) {
...     echo $bb->title . ': ' . $bb->price . ' (' .
...          $bb->superrubric_name . ' -> ' . $bb->rubric_name .
...          ")\r\n";
... }
Гараж: 300100 (Здания -> Гаражи)
Дача: 100000 (Здания -> Дачи)
Дом: 5000000 (Здания -> Дома)
Дом: 200000 (Здания -> Дома)
```

```
Жигули: 1000000 (Транспорт -> Легковой)
. . .
```

Второй формат вызова дополнительно позволяет задать условия фильтрации связываемых записей. *Анонимная функция* принимает с параметром «пустой» объект связи и собственно устанавливает связь, вызывая у полученного объекта следующие методы:

- `on()` — устанавливает связь на основе совпадения значений полей с заданными *именами*:

  ```
  on(<имя сравниваемого поля 1>, <оператор сравнения>,
     <имя сравниваемого поля 2>[, <логический оператор>='and'])
  ```

  Пример:

  ```
  // Выбираем лишь объявления с заявленной ценой более 1 млн руб.
  $bbs = Rubric::select('bbs.*', 'rubrics.name as rubric_name')
              ->join('bbs',
                     function ($join) {
                         $join->on('bbs.rubric_id', '=',
                                   'rubrics.id')
                              ->where('price', '>', 1000000);
                     }
          )->get();
  ```

  Посредством метода `on()` можно создавать и обычные условия, наподобие генерируемых методом `where()` и подобные ему. Для этого величина, которая будет сравниваться со значениями *поля 1*, передается вместо *имени сравниваемого поля 2*.

  Вызовы методов `on()` можно сцеплять друг с другом — тогда устанавливаемые ими связи будут объединяться с использованием логического оператора, указанного в последнем параметре (по умолчанию: `AND`). Пример:

  ```
  // В предыдущем примере вместо метода where() для указания
  // условия можно использовать метод on()
  . . . $join->on('bbs.rubric_id', '=', 'rubrics.id')
             ->on('price', '>', 1000000);
  ```

- `orOn()` — то же самое, что и `on()`, только объединяет устанавливаемую им связь со связью, создаваемой предыдущим вызовом метода `on()` или `orOn()`, с помощью оператора `OR`;

- ❐ `leftJoin()` — то же самое, что и `join()`, только устанавливает левую (`left`) связь. Форматы вызова:

  ```
  leftJoin(<имя связываемой таблицы>, <имя сравниваемого поля 1>,
           <оператор сравнения>, <имя сравниваемого поля 2>)
  leftJoin(<имя связываемой таблицы>, <анонимная функция>)
  ```

- ❐ `rightJoin()` — то же самое, что и `join()`, только устанавливает правую (`right`) связь. Форматы вызова те же, что и у метода `leftJoin()`;

- ❐ `crossJoin()` — то же самое, что и `join()`, только устанавливает перекрестную (`cross`) связь. Форматы вызова те же, что и у метода `leftJoin()`.

7.5.4. Использование вложенных запросов

Вложенные запросы можно использовать в Laravel:

- для создания вычисляемых полей — в массив, передаваемый методу `select()` или `addSelect()`, добавляется элемент формата:

  ```
  <имя вычисляемого поля> => <запрос, выбирающий одну запись ↳
                              с единственным полем, чье значение ↳
                              и станет значением вычисляемого поля>
  ```

  Пример:

  ```
  // Выбираем все рубрики второго уровня и у каждой - название товара
  // из последнего созданного объявления
  $rubrics = Rubric
             ::addSelect(['last_bb_title' =>
                          BB::select('title')
                            ->whereColumn('bbs.rubric_id', 'rubrics.id')
                            ->latest()->limit(1)
                         ])
             ->whereNotNull('parent_id')->get();
  ```

- для фильтрации записей — двумя способами:

  - сравнивая значение, выданное вложенным запросом, с указанным значением.

    В методе `where()` или аналогичном ему вместо имени поля указывается анонимная функция, формирующая вложенный запрос. В качестве параметра она должна принимать «пустой» объект построителя запросов. Поскольку этот объект «пуст», следует указать ему таблицу вызовом метода `from()`:

    ```
    from(<имя таблицы>[, <псевдоним таблицы>=null])
    ```

    Допускается указать *псевдоним*, по которому к таблице можно будет обратиться в формируемом вложенном запросе.

    С помощью полученного объекта анонимная функция должна строить запрос, возвращающий одну запись с единственным полем, по значению которого и будет производиться фильтрация. Пример:

    ```
    // Выбираем все рубрики, в которых есть объявления с заявленной
    // ценой более 1 млн руб.
    $rubrics = Rubric
               ::where(function ($query) {
                          $query->select('price')->from('bbs')
                             ->whereColumn('bbs.rubric_id',
                                           'rubrics.id')
                             ->orderBy('price', 'desc')->limit(1);
                       }, '>', 1000000)->get();
    ```

    Представленные далее методы построителя запросов позволяют фильтровать записи по количеству записей в результате, выданном записанным в *анонимной функции* вложенным запросом:

    - `whereExists()` — отбирает записи, у которых запрос, записанный в *анонимной функции*, выдает хотя бы одну запись:

```
whereExists(<анонимная функция>[,
            <логический оператор>='and'[,
            <инвертировать результат?>=false]])
```

Назначение параметров *логический оператор* то же, что и у метода `firstWhere()` (см. *разд. 7.3.3*). Если параметру *инвертировать результат* дать значение `true`, то метод будет выбирать записи, у которых вложенный запрос, наоборот, *не вернет ни одной* связанной записи. Пример:

```
// Выбираем все рубрики, в которых есть объявления с
// заявленной ценой более 1 млн руб.
$rubrics = Rubric
     ::whereExists(function ($query) {
                       $query->select('id')->from('bbs')
                             ->whereColumn('bbs.rubric_id',
                                           'rubrics.id')
                             ->where('price', '>', 1000000);
       })->get();
```

- `orWhereExists()` — то же самое, что и `whereExists()`, только задаваемое им условие объединяется с предыдущим посредством логического оператора `OR`:

```
orWhereExists(<анонимная функция>[,
              <инвертировать результат?>=false])
```

Пример:

```
// Выбираем все рубрики, в которых есть объявления
// с заявленной ценой более 1 млн руб. ИЛИ объявления
// о продаже "Запорожцев"
$rubrics = Rubric
     ::whereExists(function ($query) {
                       $query->select('id')->from('bbs')
                             ->whereColumn('bbs.rubric_id',
                                           'rubrics.id')
                             ->where('price', '>', 1000000);
                   })
     ->orWhereExists(function ($query) {
                       $query->select('id')->from('bbs')
                            ->whereColumn('bbs.rubric_id',
                                          'rubrics.id')
                            ->where('bbs.title', 'Запорожец');
                       })->get();
```

- `whereNotExists()` — выбирает записи, у которых запрос, записанный в *анонимной функции*, не выдает ни одной записи:

```
whereNotExists(<анонимная функция>[,
               <логический оператор>='and'])
```

- `orWhereNotExists(<анонимная функция>)` — то же самое, что и `whereNotExists()`, только задаваемое им условие объединяется с предыдущим посредством логического оператора `OR`;

- сравнивая значение указанного поля со значением, выданным вложенным запросом.

  В методе `where()` или аналогичном ему вместо сравниваемого значения указывается анонимная функция, формирующая вложенный запрос:

```
// Выбираем все объявления с заявленной ценой, превышающей
// среднюю
$bbs = Bb::where('price', '>',
                 function ($query) {
                    $query->selectRaw('avg(price)')->from('bbs');
                 }
       )->get();
```

❐ для сортировки записей — в методе `orderBy()` вместо имени поля указывается запрос, выдающий одну запись с единственным полем, по значению которого и будет выполняться сортировка:

```
// Выбираем рубрики второго уровня, отсортированные по возрастанию
// минимальной заявленной стоимости, которая указана в объявлениях
$rubrics = Rubric::whereNotNull('parent_id')
           ->orderBy(BB::select('price')
                 ->whereColumn('bbs.rubric_id', 'rubrics.id')
                 ->orderBy('price')->limit(1)
             )->get();
```

❐ для установления связи с вложенными запросами — для этого применяются следующие методы:

- `joinSub()` — устанавливает связь заданного *типа* с записями указанного *вложенного запроса*. Форматы вызова:

```
joinSub(<вложенный запрос>, <псевдоним вложенного запроса>,
        <имя сравниваемого поля 1>, <оператор сравнения>,
        <имя сравниваемого поля 2>[, <тип связи>='inner'])
joinSub(<вложенный запрос>, <псевдоним вложенного запроса>,
        <анонимная функция>)
```

  Псевдоним служит для ссылки на вложенный запрос при указании *имен полей*.

  В первом формате вызова этого метода третий и последующие параметры имеют то же назначение, что и у метода `join()` (см. *разд. 7.5.3*):

```
// Выбираем названия рубрик, в которых есть объявления
// с заявленной ценой менее 1 млн руб., и цены связанных
// с рубриками объявлений
$qb = Bb::select('price as bb_price', 'rubric_id')
        ->where('price', '<=', 1000000);
$rubrics = Rubric::select('rubrics.name', 'bb_price')
                 ->joinSub($qb, 'cbbs', 'rubrics.id', '=',
                           'cbbs.rubric_id')
           ->get();
```

  Во втором формате указывается *анонимная функция*, задающая условия фильтрации связанных записей и аналогичная таковой у метода `join()`:

```
$rubrics = Rubric::select('rubrics.name', 'bb_price')
          ->joinSub($qb, 'cbbs',
              function ($join) {
                  $join->on('rubrics.id', '=',
                            'cbbs.rubric_id');
              }
            )->get();
```

- `leftJoinSub()` — то же, что и `joinSub()`, только устанавливает левую (`left`) связь:

```
leftJoinSub(<вложенный запрос>, <псевдоним вложенного запроса>,
            <имя сравниваемого поля 1>, <оператор сравнения>,
            <имя сравниваемого поля 2>)
```

- `rightJoinSub()` — то же, что и `joinSub()`, только устанавливает правую (`right`) связь. Формат вызова такой же, как и у метода `leftJoinSub()`;
- `crossJoinSub()` — устанавливает перекрестную (`cross`) связь:

```
crossJoinSub(<вложенный запрос>, <псевдоним вложенного запроса>)
```

7.5.5. Использование фасада *DB* для выборки данных

Для выборки записей можно использовать фасад `DB`:

```
>>> use Illuminate\Support\Facades\DB;
>>> $rubrics = DB::table('rubrics')->select('name')->get();
```

Поскольку модели в таких случаях не задействуются, выборка выполняется несколько быстрее. Однако в качестве результата возвращается коллекция не объектов модели, а обычных ассоциативных массивов. Соответственно сохранять и удалять записи с ее помощью нельзя. Пример:

```
>>> // Мы можем получать значения полей записей
>>> echo $rubrics[4]->name;
Здания
>>> $rubrics[4]->name = 'Строения';
>>> // Попытка выполнить это выражение вызовет ошибку
>>> $rubrics[4]->save();
```

7.6. Агрегатные вычисления

7.6.1. Агрегатные вычисления по всем записям

Агрегатные функции реализуются следующими методами построителя запросов:

- `count([<имя поля>='*'])` — возвращает количество всех записей, кроме тех, у которых в поле с заданным *именем* хранится значение `null`. Если в качестве *имени поля* указать строку `'*'`, возвращает количество всех записей без исключения. Примеры:

```
>>> // Выводим количество всех объявлений
>>> echo Bb::count();
9
```

```
>>> // Выводим количество объявлений с заявленной ценой менее
>>> // 100 000 руб.
>>> echo Bb::where('price', '<', 100000)->count();
1
```

- `sum(<имя поля>)` — возвращает сумму значений числового поля с указанным *именем*;
- `min(<имя поля>)` — возвращает наименьшее значение поля с указанным *именем*:

```
>>> echo Bb::min('price');
10000.0
```

- `max(<имя поля>)` — возвращает наибольшее значение поля с указанным *именем*;
- `avg(<имя поля>)` — возвращает среднее арифметическое, вычисленное на основе значений поля с указанным *именем*;
- `average(<имя поля>)` — то же самое, что и `avg()`.

7.6.2. Агрегатные вычисления по группам записей

Для выполнения агрегатных вычислений по группам записей необходимая агрегатная функция записывается в вызов метода `select()` или `addSelect()` в виде фрагмента SQL-кода (см. *разд. 7.5.2*).

Кроме того, необходимо задать в формируемом запросе условия группировки записей и при необходимости фильтрации групп. Для этого применяются следующие методы построителя запросов:

- `groupBy()` — группирует записи по значениям полей с заданными *именами*:

```
groupBy(<имя поля 1>, <имя поля 2>, ... <имя поля n>)
```

Пример:

```
>>> // Вычисляем среднюю заявленную цену объявлений по рубрикам
>>> $results = Bb::select('rubric_id',
...                       DB::raw('avg("price") as avg_price'))
...              ->groupBy('rubric_id')->get();
>>> foreach ($results as $r) {
...     echo $r->rubric->name . ' - ' . $r->avg_price . "\r\n";
... }
Дома - 2600000
Гаражи - 300100
. . .
Дачи - 100000
```

- `having()` — отбирает группы, у которых сравнение содержимого поля с указанным *именем* и заданного *сравниваемого значения* выдает положительный результат:

```
having(<имя поля>[, <оператор сравнения>=null],
       <сравниваемое значение>[, <логический оператор>='and'])
```

Назначение параметра *оператор сравнения* такое же, как и у метода `firstWhere()` (см. *разд. 7.3.3*). Вызовы методов `having()` можно сцеплять друг с другом — тогда задаваемые ими условия фильтрации будут объединяться с использованием логиче-

ского оператора, указанного в последнем параметре метода (по умолчанию: AND). Пример:

```
>>> // Вычисляем среднюю заявленную цену объявлений по рубрикам и
>>> // выбираем лишь рубрики со средней ценой более 1 млн руб.
>>> $results = Bb::select('rubric_id',
...                       DB::raw('avg("price") as avg_price'))
...              ->groupBy('rubric_id')
...              ->having('avg_price', '>', 1000000)->get();
. . .
Дома - 2600000
Грузовой - 25000000
```

- orHaving() — то же самое, что и having(), только задаваемое им условие объединяется с предыдущим с использованием логического оператора OR:

```
orHaving(<имя поля>[, <оператор сравнения>=null], <сравниваемое значение>)
```

- havingBetween() — отбирает группы, у которых поле с указанным *именем* хранит значения, укладывающиеся в заданный диапазон:

```
havingBetween(<имя поля>, <массив с граничными значениями диапазона>[,
                          <логический оператор>='and'[,
                          <инвертировать результат?>=false]])
```

Массив с граничными значениями должен содержать два элемента: начальное и конечное значения диапазона. Назначение параметра *логический оператор* то же, что и у метода firstWhere(). Если параметру *инвертировать результат* дать значение true, метод будет отбирать группы, у которых значение заданного поля, наоборот, *не* входит в указанный диапазон. Пример:

```
>>> // Вычисляем среднюю заявленную цену объявлений по рубрикам и
>>> // выбираем лишь рубрики со средней ценой от 100 до 500 тыс. руб.
>>> $results = Bb::select('rubric_id',
...                       DB::raw('avg("price") as avg_price'))
...              ->groupBy('rubric_id')
...              ->havingBetween('avg_price', [100000,500000])->get();
. . .
Гаражи - 300100
Дачи - 100000
```

- havingNull() — отбирает группы, у которых поле с указанным *именем* (или все поля с именами, приведенными в *массиве*) хранит null:

```
havingNull(<имя поля>|<массив с именами полей>[,
           <логический оператор>='and'[,
           <инвертировать результат?>=false]])
```

Если параметру *инвертировать результат* присвоить значение true, метод будет отфильтровывать группы, у которых заданное поле (или все заданные поля), наоборот, хранит значение, *отличное от* null;

- havingNotNull() — отбирает группы, у которых поле с указанным *именем* (или все поля с именами, приведенными в *массиве*) хранит значение, отличное от null. В остальном подобен методу havingNull(). Формат вызова:

```
havingNotNull(<имя поля>|<массив с именами полей>[,
              <логический оператор>='and'])
```

- `orHavingNull(<имя поля>)` — то же самое, что и `havingNull()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`;
- `orHavingNotNull(<имя поля>)` — то же самое, что и `havingNotNull()`, но объединяет задаваемое им условие с предыдущим с помощью логического оператора `OR`.

7.6.3. Агрегатные вычисления по связанным записям

Еще можно получить вместе с записью первичной модели результаты агрегатных вычислений по связанным с ней записям вторичных моделей (например, количество объявлений, связанных с выбранной рубрикой). Выполнить агрегатные вычисления по связанным записям можно:

- у отдельной записи первичной таблицы — с помощью следующих методов, вызываемых непосредственно у объекта этой записи:
  - `loadCount()` — подсчитывает количество связанных записей:

    ```
    loadCount(<имя связи>|<массив с именами связей>)
    ```

    Если указать *имя связи*, в объекте первичной записи появится свойство с именем формата `<имя связи>_count`, содержащее количество связанных записей. Имя этого свойства можно изменить, для чего в вызове метода вместо *имени связи* следует указать строку формата:

    ```
    <имя связи> as <желаемое имя свойства>
    ```

    Массив с именами связей указывается, если нужно подсчитать количество либо связанных записей нескольких вторичных таблиц, либо только связанных записей, удовлетворяющих заданным условиям фильтрации. Каждый элемент такого массива должен представлять собой либо строку с именем связи, либо конструкцию формата:

    ```
    <имя связи> => <анонимная функция, фильтрующая связанные записи>
    ```

    Вместо *имени связи* можно указать строку описанного ранее формата, задающую желаемое имя свойства для хранения количества записей. *Анонимная функция* должна принимать в единственном параметре объект построителя запросов и задавать с его помощью нужные условия фильтрации.

    Метод `loadCount()` возвращает объект текущей записи.

    Примеры:

    ```
    >>> // Получаем количество объявлений о продаже грузовых
    >>> // автомобилей
    >>> $rubric = Rubric::firstWhere('name', 'Грузовой');
    >>> $rubric->loadCount('bbs');
    >>> echo $rubric->bbs_count;
    3
    >>> // То же самое, только указываем другое имя свойства,
    >>> // хранящего количество связанных записей
    ```

```
>>> $rubric->loadCount('bbs as bbcount');
>>> echo $rubric->bbcount;
2
>>> // Получаем количество объявлений о продаже грузовых
>>> // автомобилей с заявленной ценой менее 10 млн руб.
>>> $rubric->loadCount(['bbs as bbcount' =>
...                     function ($query) {
...                        $query->where('price', '<', 10000000);
...                     }]);
>>> echo $rubric->bbcount;
2
```

- `loadSum(<имя связи>, <имя поля>)` — вычисляет сумму значений поля с заданным *именем* по связанным записям. По умолчанию создает в текущем объекте записи поле с именем формата `<имя связи>_sum_<имя поля>` для хранения результата;
- `loadMin(<имя связи>, <имя поля>)` — ищет в связанных записях минимальное значение поля с заданным *именем*. По умолчанию создает в текущем объекте записи поле с именем формата `<имя связи>_min_<имя поля>` для хранения результата. Пример:

```
>>> echo $rubric->loadMin('bbs', 'price')->bbs_min_price;
1000000
```

- `loadMax(<имя связи>, <имя поля>)` — ищет в связанных записях максимальное значение поля с заданным *именем*. По умолчанию создает в текущем объекте записи поле с именем формата `<имя связи>_max_<имя поля>` для хранения результата;
- `loadAvg(<имя связи>, <имя поля>)` — вычисляет арифметическое среднее значений поля с заданным *именем* по связанным записям. По умолчанию создает в текущем объекте записи поле с именем формата `<имя связи>_avg_<имя поля>` для хранения результата;
- `loadExists(<имя связи>)` — проверяет, существуют ли связанные записи. По умолчанию создает в текущем объекте записи поле с именем формата `<имя связи>_exists` для хранения результата. Заносит в это поле значение `true`, если связанные записи существуют, и `false` — в противном случае. Пример:

```
>>> // В рубрике "Грузовой" связанные объявления существуют
>>> $rubric->loadExists('bbs')->bbs_exists;
=> true
>>> // А в рубрике "Транспорт" — нет
>>> $rubric = Rubric::firstWhere('name', 'Транспорт');
>>> $rubric->loadExists('bbs')->bbs_exists;
=> false
```

- ❒ у всех выбираемых записей первичной модели — с помощью следующих методов, вызываемых у построителя запросов и аналогичных ранее описанным методам модели:
  - `withCount()` — аналог метода `loadCount()`:

```
>>> // Выбираем все рубрики второго уровня и вычисляем количество
>>> // имеющихся в них объявлений
```

```
>>> $rubrics = Rubric::whereNotNull('parent_id')
...                   ->withCount('bbs')->get();
>>> foreach ($rubrics as $rubric) {
...     echo $rubric->name . ' - ' . $rubric->bbs_count . '   ';
... }
Дома - 2   Гаражи - 1   Легковой - 2   Грузовой - 3   Дачи - 1 ↳
Служебные - 0
>>> // То же самое, только вычисляем количество объявлений
>>> // с заявленной ценой менее 1 млн руб.
>>> $rubrics = Rubric::whereNotNull('parent_id')
...          ->withCount(['bbs' =>
...                        function ($query) {
...                          $query->where('price', '<', 1000000);
...                        }
...                       ])->get();
. . .
Дома - 1   Гаражи - 1   Легковой - 1   Грузовой - 0   Дачи - 1 ↳
Служебные - 0
```

- `withSum()` — аналоги метода `loadSum()`;
- `withMin()` — аналоги метода `loadMin()`:

```
>>> $rubrics = Rubric::whereNotNull('parent_id')
...                   ->withMin('bbs', 'price')->get();
>>> foreach ($rubrics as $rubric) {
...     echo $rubric->name . ' - ' . $rubric->bbs_min_price .
...          "\r\n";
... }
Дома - 200000
Гаражи - 300100
. . .
```

- `withMax()` — аналоги метода `loadMax()`;
- `withAvg()` — аналоги метода `loadAvg()`;
- `withExists()` — аналоги метода `loadExists()`.

7.7. Извлечение «мягко» удаленных записей

Все методы построителя запросов, рассмотренные ранее, исключают из результатов, возвращаемых SQL-запросами, записи, подвергшиеся «мягкому» удалению. Чтобы извлечь такие записи, следует использовать следующие методы построителя запросов:

- `withTrashed()` — включает в выдаваемый запросом результат также и «мягко» удаленные записи:

```
$bbs = Bb::where('price', '>', 1000000)->withTrashed()->get();
```

- `onlyTrashed()` — выдает только «мягко» удаленные записи.

7.8. Сравнение записей

Записи считаются равными, если они имеют одинаковый ключ и хранятся в одной и той же таблице одной и той же базы данных. Для проверки двух записей на равенство следует использовать следующие методы модели:

- `is(<запись>)` — возвращает `true`, если текущая и переданная в параметре записи равны, и `false` — в противном случае;
- `isNot(<запись>)` — возвращает `true`, если текущая и переданная в параметре записи, напротив, *не* равны, и `false` — в противном случае.

Примеры:

```
>>> $rubric1 = Rubric::find(8);
>>> echo $rubric1->name;
Дачи
>>> $rubric2 = Rubric::firstWhere('name', 'Дачи');
>>> $rubric1->is($rubric2);
=> true
>>> $rubric1->isNot($rubric2);
=> false
>>> $rubric3 = Rubric::find(6);
>>> $rubric1->is($rubric3);
=> false
>>> $rubric1->isNot($rubric3);
=> true
```

7.9. Получение значения заданного поля

Если из всего выданного SQL-запросом результата требуется получить лишь значение, хранящееся в определенном поле, мы извлечем его:

- если нас интересует только первая (или единственная) запись — вызовом метода `value(<имя поля>)` построителя запросов, который вернет значение поля с указанным *именем*:

  ```
  >>> echo BB::find(1)->value('title');
  Гараж
  ```

- если требуется обработать все записи — вызовом метода `pluck()` построителя запросов, который вернет коллекцию со значениями поля с *именем*, указанным в первом параметре:

  ```
  pluck(<имя поля со значениями>[, <имя поля с ключами>=null])
  ```

  Если второй параметр не указан, будет возвращена индексированная коллекция. Если же его указать, будет возвращена ассоциативная коллекция (подобная ассоциативному массиву PHP), ключи элементов которой будут взяты из поля с указанным *именем*. Примеры:

  ```
  >>> $rubric = Rubric::find(7);
  >>> $result = $rubric->bbs()->pluck('title');
  >>> foreach ($result as $r) { echo $r . '   '; }
  ЗИЛ   МАЗ   ГАЗ
  ```

```
>>> $result = $rubric->bbs()->pluck('title', 'id');
>>> foreach ($result as $k => $r) { echo $k . ': ' . $r . '   '; }
6: ЗИЛ   8: МАЗ   9: ГАЗ
```

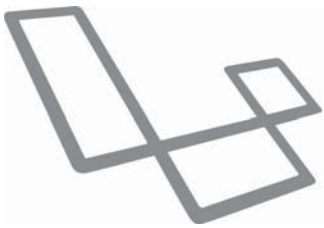
7.10. Повторное считывание записей

Выполнить повторное считывание содержимого записи из базы данных позволят следующие методы модели:

- `refresh()` — повторно считывает содержимое текущей записи из базы, при этом новые значения, занесенные в свойства модели, теряются. Возвращает объект текущей записи. Пример:

```
>>> echo $rubric->name;
Грузовой
>>> $rubric->name = 'Сельскохозайственный';
>>> $rubric->refresh();
>>> echo $rubric->name;
Грузовой
```

- `fresh()` — считывает текущую запись из базы и возвращает ее в виде нового объекта. Текущий объект не изменяется.



ГЛАВА 8

Маршрутизация

Маршрут — это объект, связывающий заданные шаблонный путь и допустимый HTTP-метод с определенным действием определенного контроллера. Каждый маршрут, написанный программистом, должен входить в один из двух *списков маршрутов*.

При получении очередного клиентского запроса подсистема *маршрутизатора* просматривает список маршрутов, сравнивая полученные из запроса путь и HTTP-метод с шаблонным путем и допустимым HTTP-методом, записанными в очередном маршруте. Если путь и метод совпадают, маршрутизатор выполняет действие контроллера, указанное в совпавшем маршруте.

Процесс выяснения, какое действие какого контроллера следует выполнить, на основании пути и HTTP-метода, извлеченных из клиентского запроса, носит название *маршрутизации*.

8.1. Настройки маршрутизатора

Настройки маршрутизатора хранятся в двух местах. Во-первых, это провайдер `App\Providers\RouteServiceProvider`, инициализирующий маршрутизатор. В его классе объявлена общедоступная константа `HOME`, задающая путь для перенаправления после успешного входа (по умолчанию: **/home** — это путь раздела пользователя).

Во-вторых, это «корневой» класс маршрутизатора `App\Http\Kernel`. Он указывает перечень посредников, связываемых с маршрутами того или иного рода. В «корневом» классе объявлены следующие защищенные свойства:

- `middleware` — массив посредников, связываемых со всеми маршрутами (это значит, что запросы и ответы, «проходящие» через все маршруты, обрабатываются этими посредниками). Каждый элемент массива представляет собой строку с полным путем к классу посредника;
- `routeMiddleware` — ассоциативный массив посредников, изначально не связанных ни с одним маршрутом, и их обозначений. Ключ элемента массива задает обозначение посредника, а значением элемента является полный путь к классу этого посредника. Пример:

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    . . .
];
```

Связать такой посредник с маршрутом можно, вызвав у последнего метод `middleware()` с указанием обозначения нужного посредника (подробности — далее).

- `middlewareGroups` — ассоциативный массив *групп посредников*. Такую группу можно связать с маршрутом, указав ее название в методе `middleware()`. Ключ элемента массива задает название группы, а значением элемента является массив с полными путями к классам посредников, входящих в группу, в виде строк или строковых обозначений посредников, заданных в массиве из свойства `routeMiddleware`.

  Изначально в массиве присутствуют группы `web` и `api`, автоматически связываемые с веб-маршрутами и API-маршрутами соответственно;

Любой из упомянутых ранее массивов можно изменять, добавляя нужных посредников и удаляя неиспользуемых. Также можно создавать новые группы посредников и удалять ненужные.

8.2. Списки маршрутов

PHP-модули со списками маршрутов хранятся в папке routes в двух файлах:

- web.php — *веб-маршруты*, ведущие на действия контроллеров, которые выдают обычные веб-страницы. С этими маршрутами связаны посредники, обеспечивающие обработку cookie, поддержку серверных сессий (и соответственно хранение данных клиента на стороне сервера) и защиту от межсайтовых запросов.

  Изначально содержит маршрут с шаблонным путем / (прямой слеш — «корень» сайта) и методом GET, выводящий страницу с приветствием, сгенерированную на основе шаблона welcome.blade.php;

- api.php — *API-маршруты*, ведущие на действия контроллеров, которые выдают данные в формате JSON, предназначенные для клиентских веб-сценариев. К шаблонным путям, записанным в этих запросах, автоматически добавляется префикс **/api**.

  Изначально содержит маршрут с шаблонным путем **/user** и методом GET, связанный с посредником `auth:sanctum` (реализует разграничение доступа) и выводящий объект текущего пользователя в формате JSON.

8.3. Создание простых маршрутов

Созданием маршрутов (точнее, представляющих их объектов класса `Illuminate\Routing\Route`) занимается подсистема маршрутизатора, управляемая посредством фасада `Illuminate\Support\Facades\Route`.

Для создания маршрутов применяются методы: `get()`, `post()`, `put()`, `patch()`, `delete()` и `options()`, возвращающие объекты маршрутов, в которых уже указан одноименный допустимый HTTP-метод (POST — при создании маршрута вызовом метода `post()`, PUT — в маршруте, созданном методом `put()`, и т. д.). Единственное исключение:

в маршруте, созданном методом `get()`, помимо HTTP-метода GET, записан еще и метод HEAD.

Формат вызова всех упомянутых ранее методов:

<метод>(*<шаблонный путь>*, *<контроллер или действие>*)

Шаблонный путь указывается в виде строки без завершающего слеша. Во втором параметре можно задать:

- непосредственно контроллер-функцию (пример показан в листинге 1.2);
- массив из двух строковых элементов — полного пути к контроллеру-классу и имени действия:

```
use App\Http\Controllers\MainController;
use App\Http\Controllers\Admin\UserController;
. . .
Route::get('/', [MainController::class, 'index']);
Route::get('/profile', [UserController::class, 'profile']);
```

- строку с путем к контроллеру-классу одного действия (будут описаны в *разд. 9.1.2.2*):

```
use App\Http\Controllers\MainPageOneActionController;
. . .
Route::get('/', MainPageOneActionController::class);
```

Могут пригодиться следующие методы фасада `Route`:

- `match()` — создает маршрут с несколькими допустимыми HTTP-методами, наименования которых приводятся в заданном *массиве*:

```
match(<массив с наименованиями HTTP-методов>, <шаблонный путь>,
      <контроллер или действие>)
```

Пример:

```
Route::match(['PUT', 'PATCH'], '/update/',
                               [BbsController::class, 'update']);
```

- `any()` — создает маршрут, у которого в качестве допустимых указаны все HTTP-методы. Формат вызова такой же, как и у методов `get()`, `post()` и др. Пример:

```
Route::any('/all', [MainController::class, 'processAll']);
```

> ***МАРШРУТИЗАТОР ПРОСМАТРИВАЕТ ПРИВЕДЕННЫЕ В СПИСКЕ МАРШРУТЫ В ТОМ ПОРЯДКЕ, В КОТОРОМ ОНИ ЗАПИСАНЫ***
>
> Как только будет найден совпадающий маршрут, просмотр списка прекращается. Поэтому, если написать два маршрута:
>
> ```
> Route::get('/', [MainController::class, 'index']);
> Route::get('/', [OtherController::class, 'index']);
> ```
>
> всегда будет совпадать первый маршрут, а второй не совпадет ни разу.

8.3.1. Специализированные маршруты

Специализированные маршруты могут самостоятельно вывести страницу или выполнить перенаправление. Они создаются следующими методами фасада `Route`:

- `view()` — создает маршрут, который при переходе по указанному *шаблонному пути* с использованием HTTP-методов GET или HEAD генерирует страницу на основе шаблона с заданным *путем* и указанного *контекста шаблона*:

  ```
  view(<шаблонный путь>, <путь к шаблону>[, <контекст шаблона>=[]]
  ```

  Пример:

  ```
  Route::view('/about', 'about');
  ```

- `redirect()` — создает маршрут, который при переходе по указанному *шаблонному пути* с использованием любого HTTP-метода выполняет перенаправление по заданному *целевому интернет-адресу* с заданным *кодом статуса* (по умолчанию: 302, временное перенаправление):

  ```
  redirect(<шаблонный путь>, <целевой интернет-адрес>[, <код статуса>=302])
  ```

  Пример:

  ```
  Route::redirect('/oldpage', '/newpage');
  ```

- `permanentRedirect()` — то же самое, что и `redirect()`, только выполняет постоянное перенаправление (с кодом статуса 301).

8.3.2. Резервный маршрут

Если путь, извлеченный из очередного клиентского запроса, не совпал ни с одним шаблонным путем из всех записанных в маршрутах, сработает реализованный во фреймворке *резервный маршрут*. Связанный с ним контроллер сгенерирует сообщение об ошибке 404 (запрашиваемая страница отсутствует).

Метод `fallback(<контроллер или действие>)` фасада `Route` позволяет связать с резервным маршрутом произвольный *контроллер или действие*:

```
// Теперь при переходе по неподдерживаемому пути будет выводиться
// главная страница сайта
Route::fallback([MainController::class, 'index']);
```

> ***По поводу метода `fallback()`***
>
> Вызов этого метода должен присутствовать в самом конце списка веб- или API-маршрутов.

8.4. Именованные маршруты

Маршруту можно дать уникальное имя, превратив его в *именованный*, для чего достаточно вызвать у представляющего его объекта метод `name(<имя маршрута>)`:

```
Route::get('/', [MainController::class, 'index'])->name('index');
Route::view('/about', 'about')->name('about');
```

После чего можно автоматически генерировать соответствующие этим маршрутам интернет-адреса, вызывая функцию-хелпер `route()` с указанием имени маршрута:

```
<a href="{{ route('index') }}">Главная страница</a> |
<a href="{{ route('about') }}">О сайте</a>
```

8.5. URL-параметры и параметризованные маршруты

В интернет-адрес, ведущий на какое-либо действие контроллера, можно поместить данные, необходимые этому действию для работы. Данные можно «прицепить» к концу пути в виде набора GET-параметров (в примере далее ключ рубрики передается в GET-параметре `rubric_id`, а ключ объявления — в GET-параметре `bb_id`):

http://localhost:8000/rubrics/bbs/?rubric_id=7&bb_id=12

или поместить непосредственно в путь — создав *URL-параметр*:

http://localhost:8000/rubrics/7/bbs/12/

Маршруты, содержащие в шаблонных путях URL-параметры, носят название *параметризованных*.

Чтобы создать URL-параметр, в шаблонный путь следует поместить его обозначение в формате `{<имя URL-параметра>}`, где *имя* должно содержать лишь буквы, цифры, символы подчеркивания и быть уникальным. Пример:

```
Route::get('/rubrics/{rubric_id}/bbs/{bb_id}', [BbController::class, 'show']);
```

Значения URL-параметров передаются контроллерам-функциям и действиям контроллеров-классов через указанные в их объявлении параметры (это простейший случай внедрения зависимостей). Присваивание значений URL-параметров параметрам функции (действия) выполняется в том порядке, в котором URL-параметры записаны в шаблонном пути. Имена параметров функции (действия) могут быть произвольными. Пример:

```
public function show($rubric_id, $bbId) { ... }
```

Можно создавать необязательные URL-параметры, которые могут отсутствовать в пути. *Имя* такого URL-параметра в его объявлении должно завершаться символом `?` (вопросительный знак). Пример:

```
Route::get('/search/{keyword?}', [MainController::class, 'search']);
```

Разумеется, соответствующий параметр функции (действия) должен быть помечен как необязательный. Пример:

```
public function search($keyword = '') { ... }
```

С параметризованными маршрутами связана одна тонкость. Рассмотрим пример:

```
Route::get('/{rubric_id}', [MainController::class, 'rubric']);
Route::view('/about', 'about');
```

При получении клиентского запроса с путем **/about** маршрутизатор посчитает совпадающим первый маршрут — с шаблонным путем **/{id}**. Попытка найти запись по ключу `about`, скорее всего, потерпит неудачу, и возникнет ошибка.

Исправить эту проблему можно, поменяв маршруты местами:

```
Route::view('/about', 'about');
Route::get('/{rubric_id}', [MainController::class, 'rubric']);
```

8.5.1. Указание правил для значений URL-параметров

Можно указать набор правил, которым должно удовлетворять значение какого-либо URL-параметра (например, значение параметра `bb`, посредством которого передается ключ объявления, должно содержать лишь цифры). В таком случае маршрут будет считаться совпавшим, только если значение URL-параметра удовлетворяет заданному правилу.

Наиболее востребованные правила для URL-параметра с заданным *именем* или URL-параметров с именами, приведенными в *массиве*, задаются с помощью ряда методов, которые, за единственным исключением, имеют следующий формат вызова:

```
<метод>(<имя URL-параметра>|<массив с именами URL-параметров>)
```

Вот эти методы:

- `whereNumber()` — значение URL-параметра должно содержать лишь цифры:

```
Route::get('/{rubric_id}', [MainController::class, 'rubric'])
      ->whereNumber('id');
Route::get('/rubrics/{rubric_id}/bbs/{bb_id}',
           [BbController::class, 'show'])
     ->whereNumber(['rubric_id', 'bb_id']);
```

- `whereAlpha()` — значение параметра должно содержать лишь буквы латиницы;
- `whereAlphaNumeric()` — значение параметра должно содержать лишь буквы латиницы и цифры;
- `whereUuid()` — значение параметра должно представлять собой универсальный уникальный идентификатор (UUID);
- `whereIn()` — значение параметра должно совпадать с одним из элементов указанного *массива допустимых значений*:

```
whereIn(<имя URL-параметра>|<массив с именами URL-параметров>,
        <массив допустимых значений>)
```

  Пример:

```
Route::get('/{kind}', [MainController::class, 'byKind'])
     ->whereIn('kind', ['buy', 'sell', 'change']);
```

Все эти методы в качестве результата возвращают текущий объект маршрута, что позволяет записывать их вызовы цепочкой:

```
Route::get('/rubrics/{rubric_slug}/bbs/{bb}', [BbController::class, 'show'])
     ->whereNumber('bb')->whereAlpha('rubric_slug');
```

Более сложное правило для URL-параметра можно записать в виде обычного регулярного выражения PHP:

- у маршрута — тогда значение URL-параметра с заданным *именем* будет проверяться на соответствие указанному *правилу* только в этом маршруте. Выполняется вызовом у объекта маршрута метода `where()`, поддерживающего два формата вызова:

```
where(<имя URL-параметра>, <правило>)
where(<ассоциативный массив правил>)
```

Второй формат вызова позволяет задать правила сразу для нескольких URL-параметров, указав их в заданном *массиве*. Ключи элементов этого *массива* зададут имена URL-параметров, а значения элементов — назначаемые им правила.

Метод `where()` возвращает текущий объект маршрута. Примеры:

```
Route::get('/{id}', [MainController::class, 'rubric'])
     ->where('id', '[0-9]+');
Route::get('/rubrics/{rubric_id}/bbs/{bb_id}',
           [BbController::class, 'show'])
     ->where(['bb' => '[0-9]+', 'rubric_slug' => '[a-z0-9_]+']);
```

- у URL-параметра — тогда его значение будет проверяться на соответствие указанному правилу во всех маршрутах. Задается в методе `boot()` провайдера `RouteServiceProvider` вызовом у фасада `Route` одного из следующих методов:

  - `pattern()` — задает правило для одного URL-параметра. Формат вызова схож с первым форматом метода `where()`. Пример:

    ```
    class RouteServiceProvider extends ServiceProvider {
        . . .
        public function boot() {
            . . .
            Route::pattern('rubric_slug', '[a-z0-9_]+');
        }
        . . .
    }
    ```

  - `patterns()` — задает правила сразу для нескольких URL-параметров. Формат вызова схож со вторым форматом метода `where()`. Пример:

    ```
    class RouteServiceProvider extends ServiceProvider {
        . . .
        public function boot() {
            . . .
            Route::patterns(['rubric_slug' => '[a-z0-9_]+',
                             'bb' => '[0-9]+']);
        }
    }
    ```

8.5.2. Внедрение моделей

Подсистема внедрения зависимостей Laravel может передавать контроллерам-функциям или действиям контроллеров-классов не ключи записей, полученные из URL-параметров, а непосредственно объекты моделей, представляющие записи с этими ключами (*внедрение моделей*). Если запись с заданным в URL-параметре ключом не найдена, по умолчанию будет возбуждено исключение `Illuminate\Database\Eloquent\ModelNotFoundException` и сгенерируется сообщение об ошибке 404 (ресурс не найден).

8.5.2.1. Неявное внедрение моделей

Laravel выполняет неявное внедрение моделей самостоятельно при соблюдении следующих условий:

- у параметра контроллера-функции или действия контроллера-класса, которому будет передан объект модели, — в качестве типа указан класс этой модели;
- имя URL-параметра — совпадает с именем параметра функции (действия).

Примеры:

```
// Маршрут
Route::get('/{rubric}', [MainController::class, 'rubric']);
. . .
// Действие контроллера. В параметре rubric окажется объект модели,
// хранящий рубрику с ключом, извлеченным из одноименного URL-параметра.
public function rubric(Rubric $rubric) { ... }

// Маршрут
Route::get('/{rubric_obj}/{bb_obj}', [MainController::class, 'bb']);
. . .
// Действие контроллера
public function bb(Rubric $rubric_obj, Bb $bb_obj) { ... }
```

По умолчанию фреймворк предполагает, что через URL-параметр передается ключ записи, и ищет запись по значению ее ключевого поля (чье имя задается свойством `primaryKey` модели, подробности — в *разд. 5.3.2*). Если через URL-параметр передается какое-либо другое значение, идентифицирующее запись (например, слаг), следует указать фреймворку, чтобы он искал запись по значению соответствующего поля. Сделать это можно:

- в маршруте — добавив к имени URL-параметра имя нужного поля через двоеточие:

```
// Передаем через URL-параметр rubric слаг рубрики и указываем Laravel
// искать нужную рубрику по полю slug
Route::get('/{rubric:slug}', [MainController::class, 'rubric']);
```

- в модели — тогда заданное в ней поле будет использовано для поиска записи этой модели во всех маршрутах. Необходимо переопределить в классе модели не принимающий параметров общедоступный (`public`) метод `getRouteKeyName()`, который в качестве результата должен возвращать строку с именем нужного поля. Пример:

```
// Модель
class Rubric extends Model {
    public function getRouteKeyName() {
        return 'slug';
    }
}
. . .
// Маршрут
Route::get('/{rubric}', [MainController::class, 'rubric']);
```

В параметризованном маршруте можно указать два URL-параметра, из которых первый хранит ключ записи первичной модели, а второй — ключ связанной записи вторичной модели или иное значение, однозначно идентифицирующее эту запись (например, слаг), — и фреймворк успешно извлечет обе записи:

```
// Передаем через URL-параметр rubric ключ рубрики, а через URL-параметр
// bb — ключ объявления, относящегося к этой рубрике
Route::get('/{rubric}/{bb}', [BbController::class, 'show']);
. . .
// Действие контроллера получит в первом параметре объект рубрики,
// а во втором — объект связанного с рубрикой объявления
public function show(Rubric $rubric, Bb $bb) { ... }
```

Чтобы несколько ускорить поиск связанной записи вторичной модели, можно дополнительно указать Laravel выполнять ее поиск только среди записей, связанных с записью первичной таблицы, идентифицированной по значению первого URL-параметра. Для этого следует выполнить одно из двух действий:

- вызвать у объекта маршрута метод `scopeBindings()`:

```
Route::get('/{rubric}/{bb}', [BbController::class, 'show'])
    ->scopeBindings();
```

- указать у второго URL-параметра имя поля, по которому будет выполняться поиск:

```
// Если URL-параметром bb передается ключ объявления…
Route::get('/{rubric}/{bb:id}', [BbController::class, 'show']);

// Если URL-параметром bb передается слаг…
Route::get('/{rubric}/{bb:slug}', [BbController::class, 'show']);
```

По умолчанию Laravel при поиске не будет считывать записи, подвергшиеся «мягкому» удалению. Если же требуется выполнить поиск и среди этих записей, необходимое предписание фреймворку можно дать вызовом у объекта маршрута метода `withTrashed()`:

```
Route::get('/{rubric}', [MainController::class, 'rubric'])->withTrashed();
```

Ранее говорилось, что при неуспешном поиске записи по умолчанию возбуждается исключение `ModelNotFoundException`. Можно указать какое-либо другое действие, выполняемое в таком случае. Для этого следует вызвать у объекта маршрута метод `missing(<`*анонимная функция*`>)`. Задаваемая в вызове анонимная функция должна принимать в качестве единственного параметра объект запроса (будет описан в *разд. 9.3*) и выполнять необходимое действие. Обычно такая функция генерирует какую-либо веб-страницу или производит перенаправление. Пример:

```
// Если запрошенная рубрика не найдена, выполняем перенаправление на главную
// страницу
Route::get('/{rubric}', [MainController::class, 'rubric'])->missing(
    function ($request) {
        return redirect()->route('main');
    }
);
```

8.5.2.2. Явное внедрение моделей

Если по какой-то причине невозможно соблюсти условия для выполнения неявного внедрения моделей или требуется искать внедряемую запись по более сложным условиям, следует прибегнуть к явному внедрению моделей. Можно:

- связать URL-параметр и модель — в этом случае при передаче ключа записи через этот URL-параметр поиск записи будет проводиться в заданной модели. Выполняется в теле метода `boot()` провайдера `RouteServiceProvider` вызовом у фасада `Route` метода `model()`:

  ```
  model(<имя URL-параметра>, <класс модели>[, <анонимная функция>=null])
  ```

  Анонимная функция вызывается, если поиски записи не увенчались успехом, должна принимать единственным параметром искомое значение и возвращать в качестве результата объект записи. Пример:

  ```
  use App\Models\Bb;
  class RouteServiceProvider extends ServiceProvider {
      . . .
      public function boot() {
          . . .
          Route::model('bb_obj', Bb::class, function ($value) {
              return new Bb(['id' => $value]);
          });
      }
  }
  . . .
  Route::get('/{bb_obj}', [MainController::class, 'bb']);
  . . .
  // У соответствующего параметра контроллера-функции или действия
  // контроллера-класса можно указать произвольное имя и необязательно
  // указывать тип
  public function bb($bb_record) { ... }
  ```

  Можно реализовать собственную логику поиска записей, для чего достаточно переопределить в модели общедоступный метод `resolveRouteBinding($value, $field = null)`. В параметре `value` передается значение, по которому будет выполняться поиск внедряемой записи, в параметре `field` — имя поля, по которому производится поиск (если `null` — следует искать по ключевому полю). Метод должен возвращать найденную запись или возбуждать исключение `ModelNotFoundException` в случае неудачного поиска. Пример:

  ```
  class Bb extends Model {
      . . .
      public function resolveRouteBinding($value, $field = null) {
          return $this->where('publish', true)
                      ->where($field ?? 'id', $value)->firstOrFail();
      }
  }
  ```

- сразу связать URL-параметр с собственной логикой поиска записи. Выполняется в методе `boot()` провайдера `RouteServiceProvider` вызовом у фасада `Route` метода `bind()`:

  ```
  bind(<имя URL-параметра>, <анонимная функция>)
  ```

  Логика поиска записи реализуется в заданной *анонимной функции*. Последняя должна принимать с параметрами значение, идентифицирующее внедряемую запись, и объ-

ект совпавшего маршрута и возвращать найденную запись. Если поиски записи не увенчались успехом, *анонимная функция* должна возбуждать исключение `ModelNotFoundException`. Пример:

```
class RouteServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Route::bind('bb', function ($value) {
            return Bb::where('publish', true)->where('id', $value)
                                             ->firstOrFail();
        });
    }
}
. . .
Route::get('/{bb}', [MainController::class, 'bb']);
. . .
public function bb($bb_object) { ... }
```

8.5.3. Внедрение перечислений

Помимо внедрения моделей, Laravel позволяет выполнять *внедрение перечислений*. В перечислении, имя которого совпадает с именем URL-параметра, ищется вариант, чей скалярный эквивалент совпадает со значением, переданным через этот URL-параметр, и найденный вариант передается контроллеру-функции или действию контроллера-класса. Пример

```
enum BbType: int {
    case BUY = 1;
    case SELL = 2;
}
. . .
Route::get('/{bbtype}', [MainController::class, 'byType']);
. . .
public function byType(BbType $type) { ... }
```

Если в перечислении не найден вариант с совпадающим скалярным эквивалентом, опять же, возбуждается исключение `ModelNotFoundException`.

8.5.4. Значения по умолчанию для URL-параметров

Любому URL-параметру можно дать произвольное значение по умолчанию. Оно будет использовано при формировании интернет-адреса вызовом функции `route()`, если этому URL-параметру не было дано значение явно.

Сначала необходимо создать свой посредник (как это делается, будет подробно описано в *главе 21*) и записать в его метод `handle()` вызов метода `defaults()` фасада `Illuminate\Support\Facades\URL`, который и даст URL-параметрам значения по умолчанию:

```
defaults(<ассоциативный массив со значениями по умолчанию>)
```

Ключи заданного *ассоциативного массива* должны соответствовать именам URL-параметров, а значения этих элементов и станут значениями этих параметров по умолчанию.

В качестве примера рассмотрим следующий маршрут:

```
Route::get('/home/{username}', [HomeController::class, 'index'])->name('home');
```

В листинге 8.1 приведен код посредника `SetDefaults`, дающего URL-параметру `username` в качестве значения по умолчанию регистрационное имя текущего пользователя.

Листинг 8.1. Пример посредника, задающего значение по умолчанию для URL-параметра

```
namespace App\Http\Middleware;
use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;
class SetDefaults {
    public function handle(Request $request, Closure $next) {
        URL::defaults(['username' => request()->user()->name]);
        return $next($request);
    }
}
```

Далее этот посредник следует занести в список из свойства `routeMiddleware` модуля `App\Http\Kernel`, дав ему какое-либо обозначение:

```
class Kernel extends HttpKernel {
    . . .
    protected $routeMiddleware = [
        . . .
        'defaults' => \App\Http\Middleware\SetDefaults::class
    ];
}
```

Наконец, этот посредник нужно связать с маршрутами, указывающими на контроллеры, в которых требуется генерировать интернет-адреса на основе маршрута с URL-параметром, у которого было задано значение по умолчанию:

```
Route::get('/', [MainController::class, 'index'])->middleware('defaults');
```

После чего в вызове функции `route()` в коде этого контроллера или шаблона, рендеринг которых выполняется из этого контроллера, можно не задавать значение для этого URL-параметра (если оно не отличается от заданного по умолчанию):

```
route('home');
```

Если интернет-адреса на основе этого маршрута нужно формировать в нескольких контроллерах, можно добавить посредник, задающий значения по умолчанию, в одну из групп — например, `web`, которая автоматически связывается со всеми веб-маршрутами:

```
class Kernel extends HttpKernel {
    . . .
```

```
    protected $middlewareGroups = [
        'web' => [
            . . .
            \App\Http\Middleware\SetDefaults::class
        ],
        . . .
    ];
    . . .
}
```

Тогда связывать этот посредник с отдельными маршрутами вызовом метода `middleware()` не нужно.

8.6. Дополнительные параметры маршрутов

Дополнительные параметры маршрутов (связанные с ними посредники, префиксы путей и др.) задаются вызовом у объекта маршрута следующих методов:

- `middleware()` — связывает с текущим маршрутом заданных *посредников* (или целую *группу посредников*), через которых станут проходить все запросы, получаемые от клиентов, и ответы, генерируемые контроллерами. Поддерживаются два формата вызова:

  ```
  middleware(<посредник или группа посредников>)
  middleware(<массив посредников и их групп>)
  ```

  Каждый *посредник* может быть указан в виде его обозначения, заданного в массиве из свойства `routeMiddleware` модуля `App\Http\Kernel`:

  ```
  Route::get('/home', [HomeController::class, 'index'])->middleware('auth');
  Route::get('/admin', [AuthController::class, 'login'])
       ->middleware(['auth', 'throttle']);
  ```

  …или в виде полного пути к его классу:

  ```
  Route::get( ... )->middleware(\App\Http\Middleware\SomeMiddleware::class);
  ```

  Каждая *группа посредников* указывается в виде ее обозначения, записанного в массиве из свойства `middlewareGroup` модуля `App\Http\Kernel`:

  ```
  Route::get( ... )->middleware('web');
  ```

- `withoutMiddleware()` — убирает из списка связанных с текущим маршрутом посредников с заданными *обозначениями*. Форматы вызова такие же, как и у метода `middleware()`. С помощью этого метода можно исключить также посредников, связываемых с маршрутами неявно. Пример:

  ```
  // Теперь запросы, проходящие через этот маршрут, не будут
  // "пропускаться" через посредник TrimStrings
  Route::get('/home', [HomeController::class, 'index'])
       ->withoutMiddleware(\App\Http\Middleware\TrimStrings::class);
  ```

- `prefix(<префикс шаблонного пути>)` — добавляет к шаблонному пути, заданному в текущем маршруте, указанный *префикс*:

```
// Теперь шаблонный путь маршрута будет выглядеть как /admin/rubrics
Route::get('/rubrics', [RubricsController::class, 'index'])
     ->prefix('admin');
```

- `domain(<допустимый домен>)` — указывает *допустимый домен*, с которого будут приходить запросы. В домене могут быть определены URL-параметры. Примеры:

```
Route::get('/admin', [AdminController::class, 'index'])
     ->domain('admin.supersite.ru');

Route::get('/home', [HomeController::class, 'index'])
     ->domain('{user}.supersite.ru');
. . .
public function index(User $user) { ... }
```

Все эти методы в качестве результата возвращают объект текущего маршрута, что позволяет сцеплять их вызовы:

```
Route::get('/rubrics', [RubricsController::class, 'index'])->name('rubrics')
     ->prefix('admin')->middleware('auth');
```

8.7. Группы маршрутов

Группа маршрутов служит для объединения произвольного количества маршрутов с целью задать у них одинаковые параметры — например, префикс шаблонного пути или связанных посредников.

Для создания группы маршрутов сначала следует вызвать непосредственно у фасада `Route` один из методов, описанных в *разд. 8.6*, после чего у возвращенного им результата вызвать метод `group()`, собственно создающий группу и поддерживающий два формата вызова:

```
group(<анонимная функция, создающая маршруты, которые входят в группу>)
group(<путь к модулю со списком маршрутов, входящих в группу>)
```

В первом формате указывается *анонимная функция*, не принимающая параметров, не возвращающая результата и создающая маршруты, которые войдут в группу:

```
// Создаем группу из двух маршрутов и связываем с ней посредник auth
Route::middleware('auth')->group(function () {
    Route::get('/home', [HomeController::class, 'index']);
    Route::get('/admin', [AdminController::class, 'index']);
});
```

Второй формат позволяет создать группу из маршрутов, объявленных в списке, который сохранен в другом модуле с указанным *путем*:

```
// Создаем группу из маршрутов, объявленных в списке routes\admin.php,
// связываем с ней посредник auth
Route::middleware('auth')->group(base_path('routes/admin.php'));
```

При формировании группы у фасада `Route` также можно вызвать следующие два метода:

- `controller(<путь к контроллеру-классу>)` — задает контроллер-класс с указанным *путем*, чьи действия будут вызываться при переходе по маршрутам, созданным в со-

ставе группы. При этом во вторых параметрах методов, создающих маршруты, следует указать лишь имена действий. Пример:

```
Route::controller(BbController::class)->group(function () {
    Route::get('/', 'index');
    Route::get('/{bb}', 'detail');
});
```

- `name(<префикс имени маршрута>)` — задает *префикс*, который будет добавлен к именам маршрутов, входящих в группу:

```
Route::name('private')->group(function () {
    // Этот маршрут получит имя private.home
    Route::get('/home', [AdminController::class, 'index'])->name('home');
    // А этот — имя private.adminpanel
    Route::get('/admin', [AdminController::class, 'index'])
        ->name('adminpanel');
});
```

> ***Примечание***
>
> Метод `namespace()`, существовавший в предыдущий версиях фреймворка, в Laravel 9 не поддерживается.

Вызовы описанных ранее методов можно записывать «цепочкой»:

```
Route::middleware('auth')->controller(AdminController::class)
    ->name('private')->group(function () {
   . . .
});
```

8.8. Маршруты на ресурсные контроллеры

Ресурсный контроллер-класс содержит все необходимые действия для реализации вывода, создания, правки и удаления каких-либо имеющихся в составе сайта ресурсов — например, объявлений (более подробно о ресурсных контроллерах будет рассказано в *главе 9*). Фасад `Route` предоставляет методы, создающие группы маршрутов, которые указывают сразу на все действия ресурсных контроллеров:

- `resource()` — создает группу маршрутов, указывающих на действия ресурсного контроллера-класса с заданным в виде строки *путем*, формируя шаблонные пути на основе указанного *имени ресурса*:

  `resource(<имя ресурса>, <путь к ресурсному контроллеру-классу>)`

  Пример:

  ```
  Route::resource('bbs', BbController::class);
  ```

  В результате будет создана группа со следующими маршрутами (записаны в формате «допустимый HTTP-метод — шаблонный путь — целевое действие контроллера-класса — имя маршрута», под ним указано назначение действия):

  - GET — **/bbs** — `index()` — `bbs.index`.

    Вывод перечня имеющихся объявлений;

- GET — **/bbs/{bb}** — `show()` — `bbs.show`.

  Вывод объявления с ключом, взятым из URL-параметра `bb`;

- GET — **/bbs/create** — `create()` — `bbs.create`.

  Создание ресурса, этап 1: вывод страницы с «пустой» веб-формой для ввода нового объявления;

- POST — **/bbs** — `store()` — `bbs.store`.

  Создание ресурса, этап 2: запись нового объявления, занесенного в веб-форму, в базу данных;

- GET — **/bbs/{bb}/edit** — `edit()` — `bbs.edit`.

  Правка объявления с ключом из URL-параметра `bb`, этап 1: вывод страницы с веб-формой, заполненной содержимым объявления, для правки;

- PUT и PATCH — **/bbs/{bb}** — `update()` — `bbs.update`.

  Правка объявления с ключом из URL-параметра `bb`, этап 2: запись в базу данных исправленного объявления;

- DELETE — **/bbs/{bb}** — `destroy()` — `bbs.destroy`.

  Удаление объявления с ключом из URL-параметра `bb`.

❐ `resources()` — создает сразу несколько групп маршрутов на ресурсные контроллеры:

```
resources(<ассоциативный массив с параметрами создаваемых групп>)
```

Ключи элементов *ассоциативного массива* зададут имена ресурсов, а значения элементов — пути к контроллерам-классам. Пример:

```
Route::resources(['bbs' => BbController::class,
                  'rubrics' => RubricController::class]);
```

❐ `apiResource()` — аналогичен `resource()`, только в создаваемой им группе присутствуют лишь маршруты, указывающие на действия: `index()`, `show()`, `store()`, `update()` и `destroy()` контроллера;

❐ `apiResources()` — аналогичен `resources()`, только в создаваемых им группах присутствуют лишь маршруты, указывающие на действия: `index()`, `show()`, `store()`, `update()` и `destroy()` контроллера.

Все эти методы в качестве результата возвращают объект класса `\Illuminate\Routing\PendingResourceRegistration`, представляющий созданную группу маршрутов. Методы `resource()` и `resources()` применяются для создания веб-маршрутов (хранящихся в модуле routes\web.php), а методы `apiResource()` и `apiResources()` — для создания API-маршрутов (хранятся в модуле routes\api.php).

Если ресурсный контроллер содержит какие-либо дополнительные действия сверх описанных ранее, маршруты, ведущие на эти действия, следует создавать *перед* созданием группы маршрутов на ресурсный контроллер. Пример создания маршрута, ведущего на дополнительное действие `popular` контроллера `BbController`:

```
Route::get('/bbs/popular ,[BbController::class, 'popular']);
Route::resource('bbs', BbController::class);
```

8.8.1. Маршруты на подчиненные ресурсные контроллеры

Подчиненные ресурсные контроллеры обрабатывают ресурсы, подчиненные другим ресурсам (например, объявления, подчиненные рубрикам). Маршруты на них можно сформировать с помощью методов, описанных в *разд. 8.8*, указав в первом параметре строку формата:

<имя родительского ресурса>.<имя подчиненного ресурса>

Например, вызов метода:

```
Route::resource('rubrics.bbs', RubricBbController::class);
```

создаст группу из следующих маршрутов (записаны в формате «допустимый HTTP-метод — шаблонный путь — целевое действие контроллера — имя маршрута»):

- GET — **/rubrics/{rubric}/bbs** — `index()` — `rubrics.bbs.index`;
- GET — **/rubrics/{rubric}/bbs/{bb}** — `show()` — `rubrics.bbs.show`;
- GET — **/rubrics/{rubric}/bbs/create** — `create()` — `rubrics.bbs.create`;
- POST — **/rubrics/{rubric}/bbs** — `store()` — `rubrics.bbs.store`;
- GET — **/rubrics/{rubric}/bbs/{bb}/edit** — `edit()` — `rubrics.bbs.edit`;
- PUT и PATCH — **/rubrics/{rubric}/bbs/{bb}** — `update()` — `rubrics.bbs.update`;
- DELETE — **/rubrics/{rubric}/bbs/{bb}** — `destroy()` — `rubrics.bbs.destroy`.

В шаблонных путях маршрутов, ведущих на действия: `show()`, `edit()`, `update()` и `destroy()`, присутствует URL-параметр `rubric` с ключом рубрики, фактически не нужный (чтобы вывести, исправить или удалить объявление, достаточно знать лишь ключ этого объявления, передаваемый в URL-параметре `bb`). Сократить шаблонные пути путем удаления из них префикса **/rubrics/{rubric}** можно вызовом у объекта созданной группы метода `shallow()`, например:

```
Route::resource('rubrics.bbs', RubricBbController::class)->shallow();
```

В результате будет создана группа со следующими маршрутами, ведущими на действия: `show()`, `edit()`, `update()` и `destroy()` (маршруты, указывающие на остальные действия, останутся такими же):

- GET — **/bbs/{bb}** — `show()` — `rubrics.bbs.show`;
- GET — **/bbs/{bb}/edit** — `edit()` — `rubrics.bbs.edit`;
- PUT и PATCH — **/bbs/{bb}** — `update()` — `rubrics.bbs.update`;
- DELETE — **/bbs/{bb}** — `destroy()` — `rubrics.bbs.destroy`.

8.8.2. Дополнительные параметры маршрутов на ресурсные контроллеры

Дополнительные параметры маршрутов на ресурсные контроллеры задаются вызовом у объекта группы, возвращенного описанными в *разд. 8.8* методами, следующих методов:

- `only()` — создает группу с маршрутами, указывающими только на действия с приведенными *именами*. Форматы вызова:

```
only(<имя действия 1>, <имя действия 2> ... <имя действия n>)
only(<массив с именами действий>)
```

Пример:

```
// Создаем маршруты только на действия index() и show()
Route::resource('bbs', BbController::class)->only('index', 'show');
```

- `except()` — создает группу с маршрутами, указывающими на все действия контроллера, кроме тех, чьи имена приведены в *массиве*. Форматы вызова:

```
except(<имя действия 1>, <имя действия 2> ... <имя действия n>)
except(<массив с именами действий>)
```

Пример:

```
// Создаем маршруты на все действия, кроме index() и show()
Route::resource('bbs', BbController::class)->except(['index', 'show']);
```

- `scoped(<массив с именами полей>)` — задает поля, по которым будет выполняться поиск внедряемых записей модели (вместо используемых по умолчанию ключевых). Ключи элементов заданного ассоциативного *массива* должны соответствовать именам URL-параметров, а значения элементов укажут имена полей модели. Пример:

```
// Указываем выполнять поиск объявлений по их слагам,
// хранящимся в поле slug модели
Route::resource('rubrics.bbs', RubricBbController::class)
     ->scoped(['bb' => 'slug']);
```

- `names(<массив с именами маршрутов>)` — задает новые имена для маршрутов. Ключи элементов заданного ассоциативного *массива* должны соответствовать действиям контроллера, а значения элементов укажут имена для соответствующих маршрутов. Пример:

```
// Даем маршруту, указывающему на действие destroy(), имя bbs.erase
Route::resource('bbs', BbController::class)
     ->names(['destroy' => 'bbs.erase']);
```

- `parameters(<массив с именами URL-параметров>)` — задает новые имена для URL-параметров. Ключи элементов заданного ассоциативного *массива* должны соответствовать именам ресурсов, а значения элементов укажут имена для соответствующих URL-параметров. Пример:

```
// Получится шаблонный путь /bbs/{bb_key}
Route::resource('bbs', BbController::class)
     ->parameters(['bbs' => 'bb_key']);;

// Получится шаблонный путь /rubrics/{rubric_key}/bbs/{bb_key}
Route::resource('rubrics.bbs', RubricBbController::class)
     ->parameters(['rubrics' => 'rubric_key', 'bbs' => 'bb_key']);
```

Также поддерживаются методы: `middleware()`, связывающий с группой маршрутов заданных посредников (подробности — в *разд. 8.6*), и `missing()`, задающий действие,

которое будет выполнено, если запись с указанным ключом не удастся найти (см. *разд. 8.5.2.1*).

8.9. Как Laravel обрабатывает списки маршрутов?

В процессе инициализации сайта фреймворк вызывает метод `boot()` провайдера `RouteServiceProvider`. В его теле присутствует вызов метода `routes()`, унаследованный от базового класса `Illuminate\Foundation\Support\Providers\RouteServiceProvider` и, собственно, создающий маршруты. В качестве параметра этому методу передается анонимная функция, в теле которой создаются две группы маршрутов (для чего используются методы, описанные в *разд. 8.6* и *8.7*):

- содержащая API-маршруты из модуля routes\api.php. С этой группой маршрутов связываются посредники из группы `api`, а еще у нее задается префикс шаблонных путей **/api**;
- содержащая веб-маршруты из модуля routes\web.php. С этой группой маршрутов связываются посредники из группы `web`.

Можно добавить в проект произвольное количество списков маршрутов. Вот пример обработки списка маршрутов из вновь созданного модуля routes\admin.php, ведущих на страницы административного раздела сайта:

```
class RouteServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        $this->routes(function () {
            . . .
            Route::middleware(['web', 'auth'])->prefix('admin')
                 ->group(base_path('routes/admin.php'));
        });
    }
    . . .
}
```

8.10. Вывод списка созданных маршрутов

Для вывода списка созданных к текущему моменту маршрутов применяется команда:

```
php artisan route:list [--sort=<обозначение колонки> [reverse]] ↳
[--method=<допустимый HTTP-метод>] [--name=<имя маршрута>] ↳
[--domain=<допустимый домен>] [--path=<шаблонный путь>] ↳
[--except-path=<шаблонный путь>] [--except-vendor] [--json]
```

Список маршрутов выводится в виде таблицы с тремя колонками:

- допустимый HTTP-метод. Если в маршруте указано несколько допустимых методов, при выводе они разделяются символами вертикальной черты;
- шаблонный путь без начального слеша;

- в зависимости от того, на что указывает маршрут:
  - если на действие контроллера-класса — строка формата:

    <имя маршрута> > <путь к контроллеру-классу>@<имя действия>

    Путь к контроллеру-классу указывается относительно пространства имен `App/Http/Controllers`;
  - если на контроллер-функцию — ничего не выводится.

По умолчанию маршруты сортируются по шаблонному пути.

Поддерживаются следующие командные ключи:

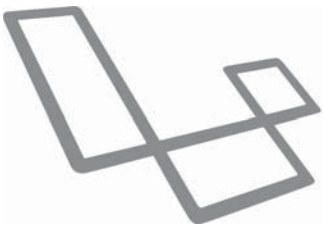
- `--sort` — сортирует список по колонке с указанным *обозначением*: `method` (допустимый HTTP-метод), `uri` (шаблонный путь), `name` (имя маршрута) или `action` (путь к контроллеру-классу и имя действия). Если указать ключ `--reverse`, порядок сортировки будет изменен на противоположный. Пример:

  ```
  php artisan route:list --sort=action
  ```

- `--method` — выводит только маршруты с заданным *допустимым HTTP-методом*:

  ```
  php artisan route:list --method=GET
  ```

- `--name` — выводит только маршруты с заданным *именем*;
- `--domain` — выводит только маршруты с заданным *допустимым доменом*;
- `--path` — выводит только маршруты с указанным *шаблонным путем*;
- `--except-path` — наоборот, выводит все маршруты, за исключением содержащих указанный *шаблонный путь*;
- `--except-vendor` — не выводит пути, которые создаются установленными дополнительными библиотеками;
- `--json` — выводит список маршрутов в формате JSON.



ГЛАВА 9

Контроллеры и действия. Обработка запросов и генерирование ответов

Контроллер в Laravel реализует функциональность отдельного раздела сайта. Например, раздел объявлений, выводящий перечень объявлений или отдельное объявление и позволяющий добавлять, править и удалять объявления, можно реализовать в одном контроллере.

Действие — это отдельная функция, выполняемая контроллером (например, вывод перечня объявлений, вывод объявления с заданным ключом и др.).

9.1. Разновидности контроллеров и особенности работы с ними

9.1.1. Контроллеры-функции

Контроллер-функция реализуется в виде обычной функции, которая записывается непосредственно в вызов метода, создающего маршрут (см. *разд. 8.3*). Фактически контроллер-функция содержит лишь одно действие. Примеры:

```
Route::get('/', function () {
    return view('index');
});
Route::get('/{bb}', function (App\Models\Bb $bb) {
    return view('detail', ['bb' => $bb]);
});
```

Преимущество контроллеров-функций — размещение их в модулях со списками маршрутов, что позволяет уменьшить фрагментацию кода. Недостаток — их удобно применять лишь в небольших сайтах и для выполнения простейших задач (наподобие вывода несложной страницы). В противном случае список маршрутов значительно увеличится в объеме и с ним станет трудно работать.

9.1.2. Контроллеры-классы

Контроллеры-классы реализуются в виде классов, а их действия — в виде методов этих классов. В качестве примера можно привести простейший контроллер-класс, показанный в *разд. 1.4*.

По соглашениям имя контроллера-класса должно оканчиваться фрагментом `Controller`. Впрочем, это не обязательно и на работу фреймворка не влияет.

По умолчанию контроллеры-классы объявляются непосредственно в пространстве имен `App\Http\Controllers` или в любом вложенном в него.

Класс контроллера является производным от базового класса `App\Http\Controllers\Controller`. Изначально последний не содержит ни свойств, ни методов и лишь использует трейты `Illuminate\Foundation\Auth\Access\AuthorizesRequests` (реализует разграничение доступа), `Illuminate\Foundation\Bus\DispatchesJobs` (позволяет заносить отложенные задания в очередь, подробности — в *главе 25*) и `Illuminate\Foundation\Validation\ValidatesRequests` (добавляет поддержку валидации). Разработчик может добавить в базовый класс контроллера любые нужные ему свойства и методы. В свою очередь, базовый контроллер является производным от класса `Illuminate\Routing\Controller`.

Применение контроллеров-классов позволяет вынести программную логику из списков маршрутов, что упрощает сопровождение сложных сайтов, но увеличивает фрагментацию кода.

9.1.2.1. Ресурсные контроллеры

Ресурсный контроллер обеспечивает функциональность вывода, создания, правки и удаления каких-либо имеющихся в составе сайта ресурсов (например, объявлений). Он реализуется как обычный контроллер-класс, содержащий набор действий со строго определенными именами. Пример ресурсного контроллера показан в листинге 9.1, назначение имеющихся в нем действий описано в комментариях к ним.

Листинг 9.1. Пример ресурсного-контроллера

```
class BbController extends Controller {
    // Вывод перечня имеющихся ресурсов
    public function index() { }

    // Вывод отдельного ресурса с ключом id
    public function show($id) { }

    // Вывод страницы с веб-формой для создания нового ресурса
    public function create() { }

    // Сохранение нового ресурса (request — это объект запроса)
    public function store(Request $request) { }

    // Вывод страницы с веб-формой для правки ресурса с ключом id
    public function edit($id) { }
```

```
    // Сохранение исправленного ресурса с ключом id
    // (request — это объект запроса)
    public function update(Request $request, $id) { }

    // Удаление ресурса с ключом id
    public function destroy($id) { }
}
```

Все маршруты, указывающие на действия ресурсного контроллера, создаются вызовом метода `resource()` фасада `Route` (см. *разд. 8.8*):

```
Route::resource('rubrics', RubricController::class);
```

Подчиненный ресурсный контроллер служит для обработки ресурсов, подчиненных другим ресурсам (например, объявлений, подчиненных рубрикам). Пример такого контроллера показан в листинге 9.2.

Листинг 9.2. Пример подчиненного ресурсного-контроллера

```
class RubricBbController extends Controller {
    // Вывод перечня имеющихся ресурсов, подчиненных ресурсу rubric
    public function index(Rubric $rubric) { }

    // Вывод отдельного ресурса bb, подчиненного ресурсу rubric
    public function show(Rubric $rubric, Bb $bb) { }

    // Вывод страницы с веб-формой для создания нового ресурса,
    // подчиненного ресурсу rubric
    public function create(Rubric $rubric) { }

    // Сохранение нового ресурса, подчиненного ресурсу rubric
    public function store(Request $request, Rubric $rubric) { }

    // Вывод страницы с веб-формой для правки ресурса bb,
    // подчиненного ресурсу rubric
    public function edit(Rubric $rubric, Bb $bb) { }

    // Сохранение исправленного ресурса bb, подчиненного ресурсу rubric
    public function update(Request $request, Rubric $rubric, Bb $bb) { }

    // Удаление ресурса bb, подчиненного ресурсу rubric
    public function destroy(Rubric $rubric, Bb $bb) { }
}
```

Создать маршруты на действия подчиненного контроллера можно вызовом того же метода `resource()` фасада `Route`:

```
Route::resource('rubrics.bbs', RubricBbController::class);
```

Ресурсные API-контроллеры отличаются от обычных тем, что не содержат действий `create()` и `edit()`.

9.1.2.2. Контроллеры одного действия

Контроллер одного действия — это контроллер-класс, который содержит лишь одно действие, реализованное в виде общедоступного метода `__invoke()`. Пример такого контроллера показан в листинге 9.3.

Листинг 9.3. Пример контроллера одного действия

```
class MainPageOneActionController extends Controller {
    public function __invoke() {
        return view('index');
    }
}
```

При создании маршрута, указывающего на такой контроллер-класс, в вызове нужного метода указывается лишь путь к контроллеру, например:

```
Route::get('/', MainPageOneActionController::class);
```

9.1.2.3. Создание контроллеров-классов

Новый контроллер-класс создается командой:

```
php artisan make:controller <имя контроллера> [--resource ↵
[--model=<имя модели> [--parent=<имя родительской модели>]] [--api] ↵
[--requests]] [--invokable] [--force]
```

По умолчанию создается обычный «пустой» контроллер без действий.

Поддерживаются следующие командные ключи:

- `--resource` — создает ресурсный контроллер. Дополнительно можно указать следующие ключи:
  - `--model` — создает ресурсный контроллер, рассчитанный на работу с моделью с указанным *именем*. В объявлениях действий этого контроллера будут подставлены параметры с именами, совпадающими с заданным *именем модели*, и *именем модели* в качестве типа, — чтобы Laravel смог выполнить внедрение модели (подробности — в *разд. 8.5.2*);
  - `--parent` — при использовании с ключом `--model` создает подчиненный ресурсный контроллер, рассчитанный на работу с родительской моделью с заданным *именем* и подчиненной моделью, которая указана в ключе `--model`;
  - `--api` — создает ресурсный API-контроллер;
  - `--requests` — дополнительно создает формальные запросы (подробнее — в *главе 10*) для использования в операциях сохранения новой и исправления существующей записи. Классы создаваемых формальных запросов будут иметь имена формата, соответственно `Store<имя класса модели>Request` и `Update<имя класса модели>Request`;
- `--invokable` — создает контроллер одного действия;
- `--force` — принудительно создает контроллер, даже если одноименный модуль уже существует.

Контроллер также можно создать одновременно с моделью, указав в команде `make:model` ключ `--controller` или `--all` (подробности — в *разд. 5.1*).

9.1.2.4. Связывание посредников с контроллерами

Посредников можно связать не только с конкретным маршрутом, но и с определенным контроллером-классом. Тогда посредники станут обрабатывать все поступающие в этот контроллер клиентские запросы и все генерируемые им ответы.

Для связывания посредников с контроллером-классом следует объявить у него конструктор и в нем вызвать унаследованный от базового класса метод `middleware()`, поддерживающий те же форматы вызова, что и одноименный метод маршрута (см. *разд. 8.6*):

```
class AdminController extends Controller {
    public function __construct() {
        $this->middleware('auth');
    }
    . . .
}
```

В качестве результата метод `middleware()` возвращает объект, хранящий параметры связанных посредников.

По умолчанию посредники будут связаны со всеми действиями контроллера. Чтобы связать их лишь с определенными действиями, можно использовать методы, вызываемые у объекта параметров посредников, который возвращается методом `middleware()`:

- `only()` — связывает посредников только с действиями с заданными *именами*. Поддерживает два формата вызова:

  ```
  only(<имя действия 1>, <имя действия 2> . . . <имя действия n>)
  only(<массив с именами действий>)
  ```

  Пример:

  ```
  public function __construct() {
      $this->middleware('auth')->only(['store', 'update', 'destroy']);
  }
  ```

- `except()` — связывает посредников со всеми действиями текущего контроллера, за исключением действий с заданными *именами*. Поддерживает те же форматы вызова, что и метод `only()`. Пример:

  ```
  public function __construct() {
      $this->middleware('auth')->except('index', 'show');
  }
  ```

9.2. Внедрение зависимостей в контроллерах

Подсистема внедрения зависимостей самостоятельно передает контроллерам-функциям и действиям контроллеров-классов значения URL-параметров, записанных в маршруте:

```
Route::get('/{id}', [MainController::class, 'rubric']);
. . .
public function rubric($id) { . . . }
```

А если какой-либо URL-параметр хранит ключ записи, то при соблюдении несложных условий (см. *разд. 8.5.2.1*) подсистема передает контроллеру непосредственно объект этой записи:

```
Route::get('/{rubric}', [MainController::class, 'rubric']);
. . .
public function rubric(Rubric $rubric) { . . . }
```

В контроллере-функции или действии контроллера-класса также можно получить объект практически любого класса, входящего в состав проекта. Нужно лишь указать в объявлении контроллера (действия) параметр и задать ему в качестве типа нужный класс. Пример:

```
use Illuminate\Http\Request;
. . .
// Получаем в параметре request объект класса Request, хранящий сведения
// о клиентском запросе
public function rubric(Request $request, Rubric $rubric) { . . . }
```

Параметры, через которые контроллер или действие получает объекты, не являющиеся записями модели, можно поставить и в конце списка параметров, используемых для получения объектов записей:

```
public function rubric(Rubric $rubric, Request $request) { . . . }
```

Однако параметры, через которые выполняется получение ключей записей (или иных значений, идентифицирующих записи, например слагов), должны располагаться строго после параметров-объектов:

```
public function rubric(Request $request, $id) { . . . }
```

9.3. Обработка клиентских запросов

Клиентский запрос представляется классом `Illuminate\Http\Request`. Получить объект этого класса можно одним из следующих способов:

- посредством внедрения зависимостей (см. *разд. 9.2*):

```
use Illuminate\Http\Request;
. . .
public function store(Request $request) {
    $title = $request->input('title');
    . . .
}
```

- вызвав функцию-хелпер `request()` без параметров:

```
$title = request()->input('title');
```

- воспользовавшись фасадом `Illuminate\Support\Facades\Request`, за которым «скрывается» подсистема обработки запросов:

```
use Illuminate\Support\Facades\Request;
. . .
$title = Request::input('title');
```

9.3.1. Извлечение данных, отправленных посетителем

Извлечь из клиентского запроса данные, отправленные посетителем из веб-формы, можно несколькими способами.

Проще всего извлечь значение, переданное в GET- или POST-параметре, обратившись к свойству объекта запроса, чье имя совпадает с именем нужного параметра:

```
// Получаем объект запроса
$request = request();
// Берем значение POST-параметра title из одноименного свойства запроса
$title = $request->title;
```

При обращении к такому свойству Laravel сначала ищет в запросе одноименный POST-параметр, в случае неуспеха — одноименный GET-параметр, далее — URL-параметр с таким же именем (о URL-параметрах рассказывалось в *разд. 8.5*). Если поиски подходящего значения не увенчаются успехом, возвращается `null`.

Еще класс запроса предоставляет следующие полезные методы:

- `boolean()` — возвращает `true`, если GET- или POST-параметр с заданным *именем* хранит значения: `1`, `'1'`, `true`, `'true'`, `'on'` или `'yes'`, и `false` — в противном случае. Используется для проверки, был ли установлен присутствующий в веб-форме флажок. Формат вызова:

  ```
  boolean(<имя параметра>[, <значение по умолчанию>=false])
  ```

  Пример:

  ```
  if ($request->boolean('agreed')) {
      // Флажок agreed установлен
  }
  ```

- `date()` — возвращает значение даты, переданное через GET- или POST-параметр с заданным *именем*, в виде объекта класса `Carbon`, или `null`, если параметра с таким *именем* нет. Формат вызова:

  ```
  date(<имя параметра>[, <формат даты>=null[, <временна́я зона>=null]])
  ```

  Если *формат даты* не указан, будет использоваться формат:

  ```
  <год>-<номер месяца>-<число>[ <часы>:<минуты>[:<секунды>]]
  ```

  Если *временна́я зона* не указана, будет использоваться таковая, заданная в рабочей настройке `app.timezone`. Пример:

  ```
  $creationDate = $request->date('created');
  ```

  Если исходное значение даты не удалось преобразовать, возбуждается исключение `InvalidArgumentException`, встроенное в PHP;

- `route()` — возвращает значение URL-параметра с указанным *именем*:

  ```
  route(<имя URL-параметра>[, <значение по умолчанию>=null])
  ```

Если URL-параметр с заданным *именем* отсутствует, возвращается *значение по умолчанию*;

- `input()` — в зависимости от формата вызова:
  - `input(<имя параметра>[, <значение по умолчанию>=null])`

    Возвращает значение GET- или POST-параметра с указанным *именем* или, если такой параметр в запросе отсутствует, — заданное *значение по умолчанию*:

    ```
    $price = $request->input('price', '');
    ```

    С той же целью можно использовать функцию `request()`, которая в этом случае вызывается в таком же формате:

    ```
    $price = request('price', '');
    ```

    Если с каким-либо из параметров передается массив, обычный или ассоциативный, для доступа к элементам этого массива следует применять «точечную» нотацию:

    ```
    $address0City = $request->input('addresses.0.city');
    $address0Street = $request->input('addresses.0.street');
    $address1City = $request->input('addresses.1.city');
    . . .
    ```

  - `input()` — возвращает ассоциативный массив со значениями всех GET- и POST-параметров, что есть в запросе. Ключи элементов этого массива будут иметь те же имена, что и параметры. Пример:

    ```
    $allParams = Request::input();
    $price = $allParams['price'];
    ```

- `query()` — аналогичен `input()`, но работает только с GET-параметрами:

  ```
  $search = Request::query('search');
  ```

- `all()` — в зависимости от формата вызова:
  - `all()` — то же самое, что и метод `input()`, вызванный без параметров;

    ```
    $allValues = $request->all();
    $title = $allValues['title'];
    ```

  - `all(<имя параметра 1>, <имя параметра 2> . . . <имя параметра n>)`
    `all(<массив с именами параметров>)`

    Возвращает ассоциативный массив, содержащий значения GET- и POST-параметров с указанными *именами*:

    ```
    $someValues = Request::all('title', 'content', 'address');
    $address = $someValues['address'];
    ```

    С той же целью можно использовать функцию `request()`, которая в этом случае вызывается с указанием *массива с именами* в качестве единственного параметра:

    ```
    $someValues = request(['title', 'content', 'address']);
    ```

- `only()` — аналогичен второму формату вызова метода `all()`:

  ```
  $someValues = Request::only('title', 'content', 'address');
  ```

- `except()` — возвращает ассоциативный массив со значениями всех GET- и POST-параметров, кроме имеющих указанные *имена*. Поддерживает два формата вызова:

  ```
  except(<имя параметра 1>, <имя параметра 2> . . . <имя параметра n>)
  except(<массив с именами параметров>)
  ```

  Пример:

  ```
  $someValues = $request->except(['rubruc_id', 'published']);
  ```

- `collect()` — возвращает коллекцию `Collection`, содержащую, в зависимости от формата вызова:

  - `collect(<массив с именами параметров>)` — элементы со значениями всех параметров с перечисленными в *массиве* именами:

    ```
    $cRubric = $request->collect(['name', 'parent_id']);
    $rubricName = $cRubric['name'];
    ```

  - `collect(<имя параметра>)` — единственный элемент со значением параметра с заданным *именем*;

  - `collect()` — элементы со значениями всех GET- и POST-параметров, что есть в запросе.

    ВСЕ ЗНАЧЕНИЯ, ПЕРЕДАВАЕМЫЕ В СОСТАВЕ ЗАПРОСА…

    …пропускаются через посредников `TrimStrings` и `ConvertEmptyStringsToNull`. Первый посредник удаляет из значений начальные и конечные пробелы, второй преобразует «пустые» строки в значения `null`.

9.3.2. Как узнать, присутствует ли в запросе нужное значение?

Выяснить, присутствует ли в полученном запросе GET- или POST-параметр с заданным *именем*, позволят следующие методы, поддерживаемые классом запроса:

- `has()` — в зависимости от формата вызова:

  - `has(<имя параметра>)` — возвращает `true`, если в текущем запросе есть GET- или POST-параметр с заданным *именем*, и `false` — в противном случае:

    ```
    if (Request::has('address')) {
        // Посетитель ввел адрес
    }
    ```

  - `has(<имя параметра 1>, <имя параметра 2> . . . <имя параметра n>)`
    `has(<массив с именами параметров>)`

    Возвращает `true`, если в текущем запросе есть *все* GET- или POST-параметры с заданными *именами*, и `false` — в противном случае;

- `exists()` — то же самое, что и `has()`;

- `hasAny()` — возвращает `true`, если в текущем запросе есть *хотя бы один* GET- или POST-параметр с заданным *именем*, и `false` — в противном случае. Формат вызова аналогичен второму формату метода `has()`. Пример:

```
if (Request::hasAny('title', 'content', 'price')) {
    . . .
}
```

- `missing()` — возвращает `true`, если в текущем запросе, наоборот, отсутствует GET- или POST-параметр (параметры) с заданным *именем* (*именами*), и `false` — в противном случае. Форматы вызова аналогичны таковым у метода `has()`. Примеры:

```
if ($request->missing(['content', 'address'])) {
    // Посетитель не ввел ни содержания объявления, ни адреса
}
```

- `whenHas()` — если в запросе есть GET- или POST-параметр с заданным *именем*, вызывает *анонимную функцию 1*, передавая ей значение этого параметра. Если *функция 1* возвращает какой-либо результат, метод возвращает его, в противном случае возвращает текущий объект запроса.

  Если же в запросе нет такого параметра, вызывает не принимающую параметров *анонимную функцию 2* и возвращает результат ее выполнения. Если *функция 2* не задана, возвращает текущий объект запроса.

  Формат вызова:

```
whenHas(<имя параметра>, <анонимная функция 1>[,
                         <анонимная функция 2>=null])
```

  Пример:

```
$rubricName = $request->whenHas('name', function ($name) {
    return ucfirst($name);
}, function () {
    return Rubric::first();
});
```

- `filled()` — аналогичен `has()`, но возвращает `true` только в том случае, если параметры с заданными *именами* еще и не «пусты»;
- `anyFilled()` — аналогичен `hasAny()`, но возвращает `true` только в том случае, если параметры с заданными *именами* еще и не «пусты»;
- `isNotFilled()` — аналогичен `filled()`, только возвращает `true` в том случае, если параметры с заданными *именами*, наоборот, «пусты»;
- `whenFilled()` — аналогичен `whenHas()`, только проверяет, чтобы параметр с заданным именем еще и не был «пуст».

9.3.3. Добавление в запрос произвольных значений

Иногда может потребоваться добавить в состав данных, полученных в запросе через GET- и POST-параметры, какие-либо дополнительные значения. Для этого класс запроса поддерживает следующие методы:

- `merge(<ассоциативный массив>)` — добавляет в состав значений, полученных в текущем запросе, значения, приведенные в заданном *ассоциативном массиве*. Ключи элементов этого *массива* зададут имена у добавляемых значений. Если в текущем запросе уже существует значение с заданным именем, оно будет перезаписано. Пример:

```
use Illuminate\Support\Str;
. . .
// Добавляем в запрос значение user, хранящее ключ текущего пользователя
$request->merge(['user' => Auth::user()->id]);
```

- `mergeIfMissing()` — аналогичен `merge()`, только добавляет в запрос лишь отсутствующие в нем значения, оставляя существующие без изменений.

9.3.4. Получение сведений о запросе

Для получения различных сведений о клиентском запросе применяются следующие методы, поддерживаемые классом запроса:

- `secure()` — возвращает `true`, если текущий запрос был выполнен по протоколу HTTPS, и `false` — если был выполнен по протоколу HTTP;
- `ajax()` — возвращает `true`, если был выполнен AJAX-запрос, и `false` — в противном случае;
- `pjax()` — возвращает `true`, если был выполнен PJAX-запрос, и `false` — в противном случае;
- `expectsJson()` — возвращает `true`, если клиент запрашивает данные в формате JSON, и `false` — в противном случае;
- `prefetch()` — возвращает `true`, если текущий запрос является запросом на предварительную загрузку файла, и `false` — в противном случае;
- `method()` — возвращает обозначение HTTP-метода, с помощью которого был выполнен запрос;
- `isMethod(<обозначение HTTP-метода>)` — возвращает `true`, если текущий запрос был выполнен с помощью HTTP-метода с указанным обозначением, и `false` — в противном случае:

```
if (request()->isMethod('post')) {
    // Запрос был выполнен методом POST
}
```

- `isMethodSafe()` — возвращает `true`, если текущий запрос был выполнен с применением HTTP-метода: GET, HEAD, OPTIONS или TRACE, и `false` — в противном случае;
- `path()` — возвращает путь, по которому был выполнен запрос, без начального и конечного слешей. Если был выполнен запрос к «корню» сайта, будет возвращен символ прямого слеша. Примеры:

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->path();                    // Результат: rubrics/2
// Запрос по интернет-адресу http://www.supersite.ru/?search=Дом
$path = request()->path();                    // Результат: /
```

- `is()` — возвращает `true`, если путь, по которому был выполнен запрос, совпадает с одним из заданных *шаблонов*, и `false` — в противном случае. В *шаблонах* для обозначения произвольных сегментов пути можно использовать литерал `*`. Поддерживаются два формата вызова:

```
is(<шаблон 1>, <шаблон 2> . . . <шаблон n>)
is(<массив с шаблонами>)
```

Примеры:

```
// Запрос по интернет-адресу http://localhost/rubrics/create/
$isMatch = request()->is('rubrics/create');        // Результат: true
$isMatch = request()->is('rubrics/edit');          // Результат: false
// Запрос по интернет-адресу http://localhost/bbs/2/edit/
$isMatch = request()->is('bbs/*/edit');            // Результат: true
```

- `routeIs()` — возвращает `true`, если полученный клиентский запрос пришел через один из маршрутов с заданными *именами*, и `false` — в противном случае. В *именах* для обозначения последовательностей произвольных символов можно использовать литерал `*`. Форматы вызова:

```
routeIs(<имя маршрута 1>, <имя маршрута 2> . . . <имя маршрута n>)
routeIs(<массив с именами маршрутов>)
```

Примеры:

```
if ($request->routeIs('index')) {
    // Был выполнен запрос к "корню" сайта
}

if ($request->routeIs('bbs.store', 'bbs.update')) {
    // Была выполнена попытка сохранить объявление: новое или
    // уже существующее
}
```

- `root()` — возвращает интернет-адрес «корня» сайта:

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->root();              // Результат: http://localhost
```

- `url()` — возвращает интернет-адрес, по которому был выполнен текущий запрос, без GET-параметров:

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->url();     // Результат: http://localhost/rubrics/2
// Запрос по интернет-адресу http://www.supersite.ru/?search=Дом
$path = request()->url();     // Результат: http://www.supersite.ru
```

- `fullUrl()` — возвращает полный, с GET-параметрами, интернет-адрес, по которому был выполнен текущий запрос:

```
// Запрос по интернет-адресу http://www.supersite.ru/?search=Дом
$path = request()->fullUrl();
                    // Результат: http://www.supersite.ru/?search=Дом
```

- `fullUrlIs()` — возвращает `true`, если полный интернет-адрес, по которому был выполнен текущий запрос, совпадает с одним из заданных *шаблонов*, и `false` — в противном случае. В *шаблонах* для обозначения произвольных сегментов адреса можно использовать литерал `*`. Поддерживаются два формата вызова:

```
fullUrlIs(<шаблон 1>, <шаблон 2> . . . <шаблон n>)
fullUrlIs(<массив с шаблонами>)
```

Примеры:

```
// Запрос по интернет-адресу http://localhost/rubrics/create/
$isMatch = request()->fullUrlIs('*rubrics*');      // Результат: true
$isMatch = request()->fullUrlIs('http://localhost:1234/*');
                                                   // Результат: false
```

- `getScheme()` — возвращает обозначение протокола, по которому был выполнен запрос, в виде строки `'http'` или `'https'`;
- `getHost()` — возвращает интернет-адрес хоста без обозначения протокола и номера порта:

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->getHost();                  // Результат: localhost
// Запрос по интернет-адресу http://localhost:8000/rubrics/2/
$path = request()->getHost();                  // Результат: localhost
```

- `getPort()` — возвращает номер TCP-порта, через который был выполнен текущий запрос;
- `getHttpHost()` — возвращает интернет-адрес хоста с указанием номера TCP-порта, если использовался нестандартный порт;

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->getHttpHost();         // Результат: localhost
// Запрос по интернет-адресу http://localhost:8000/rubrics/2/
$path = request()->getHttpHost();         // Результат: localhost:8000
```

- `getSchemeAndHttpHost()` — возвращает полный интернет-адрес хоста, включающий обозначение протокола;
- `ip()` — возвращает IP-адрес клиента, выполнившего текущий запрос;
- `ips()` — возвращает массив IP-адресов маршрутизаторов и сетевых шлюзов, через которые прошел текущий запрос. IP-адрес отправившего его клиента будет самым последним в этом массиве;
- `segment()` — возвращает сегмент пути с заданным *номером* (нумерация сегментов начинается с 1). Если сегмента с таким *номером* нет, возвращается *значение по умолчанию*. Формат вызова:

```
segment(<номер сегмента>[, <значение по умолчанию>=null])
```

Пример:

```
// Запрос по интернет-адресу http://localhost/rubrics/2/
$path = request()->segment(1);                 // Результат: rubrics
$path = request()->segment(2);                 // Результат: 2
$path = request()->segment(3);                 // Результат: null
```

- `segments()` — возвращает массив со всеми сегментами пути;
- `userAgent()` — возвращает значение заголовка `User-Agent`;
- `getPreferredFormat()` — возвращает строку с обозначением предпочтительного формата, извлеченным из текущего запроса (например, `'html'`). Берется из первого элемента списка, приведенного в заголовке `Accept` запроса;

- `accepts()` — возвращает `true`, если клиент запрашивает данные в формате, соответствующем одному из указанных *обозначений*, и `false` — в противном случае. Форматы вызова:

  ```
  accepts(<обозначение 1>, <обозначение 2> . . . <обозначение 3>)
  accepts(<массив с обозначениями>)
  ```

  Пример:

  ```
  if (request()->accepts('text/html', 'text/plain'))
      . . .
  ```

- `getAcceptableContentTypes()` — возвращает массив с обозначениями всех форматов данных, поддерживаемых веб-обозревателем, который прислал текущий запрос (берется из заголовка `Accept` запроса);

- `getLanguages()` — возвращает массив с обозначениями всех языков, поддерживаемых веб-обозревателем, который прислал текущий запрос (берется из заголовка `Accept-Language` запроса);

- `getPreferredLanguage()` — возвращает строку с обозначением предпочтительного языка, извлеченным из текущего запроса (например, `'ru_RU'`, `'en_US'`). Берется из первого элемента списка, приведенного в заголовке `Accept-Language` запроса;

- `getCharsets()` — возвращает массив с обозначениями всех текстовых кодировок, поддерживаемых веб-обозревателем, который отправил текущий запрос (берется из заголовка `Accept-Charset` запроса);

- `getEncodings()` — возвращает массив с обозначениями всех форматов сжатия, поддерживаемых веб-обозревателем, который отправил текущий запрос (берется из заголовка `Accept-Encoding` запроса);

- `header()` — в зависимости от формата вызова:

  - `header(<имя заголовка>[, <значение по умолчанию>=null])`

    Возвращает значение заголовка с указанным *именем*, извлеченного из текущего запроса, или *значение по умолчанию*, если такой заголовок отсутствует:

    ```
    $h = request->header('Cache-Control');
    ```

  - `header()` — возвращает ассоциативный массив со всеми заголовками текущего запроса;

- `hasHeader(<имя заголовка>)` — возвращает `true`, если в текущем запросе присутствует заголовок с указанным *именем*, и `false` — в противном случае;

- `getProtocolVersion()` — возвращает строку с версией протокола HTTP в формате `HTTP/<номер версии>`.

Часть информации о клиентском запросе можно получить из следующих свойств:

- `headers` — все заголовки текущего запроса в виде объекта особого класса. Для получения значения заголовка с заданным *именем* следует вызвать у этого объекта метод `get(<имя заголовка>)`. Пример:

  ```
  $h = request()->headers->get('Cache-Control');
  ```

Примечание

Это свойство использовалось для получения заголовков клиентского запроса в предыдущих версиях Laravel.

- `server` — сведения о веб-сервере и среде исполнения, полученные из массива `$_SERVER`, в виде объекта аналогичного класса:

```
$s = request()->server->get('SERVER_SOFTWARE');
```

9.4. Генерирование интернет-адресов

Сгенерировать интернет-адрес, указывающий на текущий хост, можно на основе:

- маршрута с заданным *именем* — вызвав функцию-хелпер `route()`:

```
route(<имя маршрута>[,
      <ассоциативный массив со значениями URL-параметров>=[][,
      <полный интернет-адрес?>=true]])
```

В заданном *ассоциативном массиве* ключи элементов должны соответствовать именам URL-параметров, а значения элементов зададут значения этих параметров. Если параметру *полный интернет-адрес* дать значение `true`, будет возвращен полный интернет-адрес, если дать `false` — сокращенный. Пример:

```
$url = route('rubric', ['rubric' => 2]);
                                // Результат: http://localhost:8000/2
$url = route('rubric', ['rubric' => 2], false);
                                // Результат: /2
```

В *ассоциативном массиве* в качестве значения URL-параметра вместо ключа записи:

```
$url = route('rubric', ['rubric' => $rubric->id]);
```

можно передать объект записи — фреймворк сам извлечет из него ключ:

```
$url = route('rubric', ['rubric' => $rubric]);
```

Если из записи следует извлекать не ключ, а значение какого-либо иного поля, следует переопределить в модели общедоступный метод `getRouteKeyName()`. Он не должен принимать параметров, а в качестве результата должен возвращать имя нужного поля. Пример:

```
class Rubric extends Model {
    . . .
    public function getRouteKeyName() {
        return 'slug';
    }
}
```

Также можно непосредственно указать значение, которое должно подставляться вместо URL-параметра. Для этого достаточно переопределить в модели общедоступный метод `getRouteKey()`. Он не должен принимать параметров и должен возвращать требуемое значение. Пример:

```
class Rubric extends Model {
    . . .
```

```
    public function getRouteKey() {
        return $this->slug;
    }
}
```

Если в *ассоциативном массиве* указать элементы, не соответствующие записанным в маршруте URL-параметрам, они будут помещены в сгенерированный интернет-адрес в виде GET-параметров:

```
$url = route('rubric', ['rubric' => 2, 'anchor' => 'abc']);
                        // Результат: http://localhost:8000/2?anchor=abc
```

- действия контроллера-класса с заданным *именем* — вызвав функцию-хелпер `action()`:

```
action(<контроллер и действие>[,
       <ассоциативный массив со значениями URL-параметров>=[][,
       <полный интернет-адрес?>=true]])
```

В качестве первого параметра передается массив из двух строковых элементов: полного пути к контроллеру-классу и имени действия:

```
use App\Http\Controllers\MainController;
. . .
$url = action([MainController::class, 'index']);
```

При вызове этой функции фреймворк просматривает список маршрутов и генерирует интернет-адрес на основе первого маршрута, в котором записаны указанные *контроллер и действие* и который имеет в шаблонном пути заданные в массиве *URL-параметры*;

- произвольного *пути* — вызвав функцию-хелпер `url()`:

```
url(<путь>[, <массив с дополнительными сегментами>=[][, <HTTPS?>=null]])
```

Дополнительные сегменты, приведенные в заданном *массиве*, будут добавлены к *пути*. Если параметру *HTTPS* дать значение `true`, будет сгенерирован интернет-адрес, использующий протокол HTTPS, а если дать значение `false` — интернет-адрес с протоколом HTTP, если `null` — интернет-адрес с текущим протоколом.

Функция возвращает полный интернет-адрес, включающий адрес текущего хоста и обозначение протокола. Примеры:

```
$url = url('rubrics/2/3');
                        // Результат: http://localhost:8000/rubrics/2/3
$url = url('rubrics', [2, 3]);
                        // Результат: http://localhost:8000/rubrics/2/3
$url = url('rubrics', [2, 3], true);
                        // Результат: https://localhost:8000/rubrics/2/3
```

Функция-хелпер `secure_url()` генерирует интернет-адрес, использующий протокол HTTPS:

```
secure_url(<путь>[, <массив с дополнительными сегментами>=[]])
```

Пример:

```
$url = secure_url('rubrics', [2, 3]);
                        // Результат: https://localhost:8000/rubrics/2/3
```

Сгенерировать интернет-адрес текущей или предыдущей страницы можно, использовав объект генератора интернет-адресов. Его можно получить двумя способами:

- вызвав функцию `url()` без параметров;
- обратившись к фасаду `Illuminate\Support\Facades\URL`.

Вот методы, поддерживаемые генератором адресов:

- `current()` — возвращает текущий интернет-адрес без GET-параметров:

```
$url = url()->current();
```

- `full()` — возвращает текущий интернет-адрес с GET-параметрами;
- `previous([<запасной интернет-адрес>=false])` — возвращает интернет-адрес предыдущей страницы, извлеченный из заголовка `Referrer` или, если такого заголовка в клиентском запросе нет, из серверной сессии. Если адреса предыдущей страницы нет и в сессии, возвращается заданный *запасной интернет-адрес*, а если он не указан — адрес «корня» сайта. Пример:

```
use Illuminate\Support\Facades\URL;
$url = URL::previous('/home');
```

9.5. Генерирование серверных ответов

9.5.1. Ответы на основе шаблонов

9.5.1.1. Ответы в виде объектов класса *View*

Проще всего сгенерировать ответ на основе шаблона с заданным *путем* и указанного *контекста шаблона*, вызвав функцию-хелпер `view()`:

```
view(<путь к шаблону>[, <контекст шаблона>=[]])
```

Путь к шаблону указывается относительно папок, в которых хранятся шаблоны (по умолчанию: единственная папка resources\views). Имя шаблона в *пути* записывается без расширения blade.php. *Контекст шаблона* задается в виде ассоциативного массива, в котором ключи элементов представят переменные, создаваемые в шаблоне, а значения элементов зададут значения этих переменных.

В качестве результата возвращается объект класса `Illuminate\View\View`, представляющий подготовленный к рендерингу шаблон. Его необходимо вернуть из контроллера в качестве результата. Пример:

```
public function rubric(Rubric $rubric) {
    return view('rubric', ['rubruc' => $rubric,
                           'bbs' => $rubric->bbs()->get()]);
}
```

Разделять имена файлов и папок в *пути к шаблону* можно как слешами:

```
return view('rubrics/create');
```

так и точками (в этом случае имена файлов и папок, присутствующих в *пути*, не должны содержать точек):

```
return view('rubrics.create');
```

Рендеринг шаблона, представляемого объектом класса `View`, будет выполнен позже, непосредственно перед отправкой клиенту (*отложенный рендеринг*).

Класс `View` поддерживает следующие полезные методы:

- `with()` — добавляет в контекст шаблона новые переменные. Поддерживаются два формата вызова:

  ```
  with(<имя переменной>, <значение переменной>)
  with(<ассоциативный массив с добавляемыми переменными>)
  ```

  Примеры:

  ```
  $view = view('rubric');
  $view->with('bbs', $rubric->bbs()->get());
  return $view->with(['rubric' => $rubric, 'title' => 'Рубрики']);
  ```

- `toHtml()` — возвращает строку с HTML-кодом страницы, сгенерированной на основе текущего шаблона.

Вызвав функцию `view()` без параметров или обратившись к фасаду `Illuminate\Support\Facades\View`, можно получить объект класса `Illuminate\View\Factory`, который представляет подсистему рендеринга шаблонов. Он поддерживает три полезных метода:

- `make()` — полностью аналогичен функции `view()`, вызванной с параметрами:

  ```
  use Illuminate\Support\Facades\View;
  . . .
  return View::make('index');
  ```

- `first()` — генерирует ответ на основе первого найденного шаблона из присутствующих в *массиве*:

  ```
  first(<массив с путями к шаблонам>[, <контекст шаблона>=[]])
  ```

  Если ни один из шаблонов, присутствующих в *массиве*, не был найден, будет возбуждено исключение `InvalidArgumentException`, поддерживаемое PHP. Пример:

  ```
  return view()->first(['contacts/map', 'contacts', 'about']);
  ```

- `exists(<путь к шаблону>)` — возвращает `true`, если шаблон с заданным *путем* существует, и `false` — в противном случае:

  ```
  if (View::exists('contacts/map')) {
      return View::make('contacts/map');
  }
  ```

9.5.1.2. Ответы в виде объектов класса *Response*

Еще можно сгенерировать на основе шаблона ответ, представляемый объектом класса `Illuminate\Http\Response`. Такой ответ имеет две особенности: во-первых, рендеринг выполняется в момент генерирования ответа, а не перед его отправкой (как у ответа, представленного объектом класса `View`, см. *разд. 9.5.1.1*), во-вторых, у такого ответа можно задать дополнительные параметры.

Сначала необходимо получить объект класса `Illuminate\Routing\ResponseFactory`, посредством которого генерируются ответы такого рода. Для этого следует либо вызвать

функцию `response()` без параметров, либо обратиться к фасаду `Illuminate\Support\Facades\Response`.

Собственно генерирование ответа производит метод `view()`:

```
view(<путь к шаблону>|<массив путей к шаблонам>[, <контекст шаблона>=[][,
     <код статуса>=200[,
     <ассоциативный массив с добавляемыми в ответ заголовками>=[]]]])
```

Если в первом параметре указать *массив путей к шаблонам*, ответ будет сгенерирован на основе первого найденного шаблона из присутствующих в *массиве*. В *ассоциативном массиве* ключи элементов должны соответствовать именам заголовков, а значения элементов зададут значения для этих заголовков. Пример:

```
public function rubric(Rubric $rubric) {
    return response()
           ->view('rubric', ['rubruc' => $rubric,
                             'bbs' => $rubric->bbs()->get()],
                  200, ['Cache-Control' => 'no-cache']);
}
```

9.5.2. Специальные ответы

9.5.2.1. Отправка файла для отображения в веб-обозревателе

Чтобы сгенерировать ответ с файлом, который должен быть отображен непосредственно веб-обозревателем, следует вызвать метод `file()` класса `ResponseFactory`:

```
file(<путь к отправляемому файлу>[,
     <ассоциативный массив с добавляемыми в ответ заголовками>=[]])
```

В качестве результата возвращается объект класса `Symfony\Component\HttpFoundation\BinaryFileResponse`, представляющий ответ с выводимым файлом. Пример:

```
use Illuminate\Support\Facades\Response;
. . .
return Response::file('c:/sites/common/logo.jpg');
```

Объект, представляющий ответ с выводимым файлом, поддерживает следующие методы:

- `deleteFileAfterSend()` — предписывает удалить файл после отправки. Может пригодиться при отправке файлов, генерируемых программно и не предназначенных для длительного хранения. Пример:

  ```
  return Response::file($generatedFileName)->deleteFileAfterSend();
  ```

- `trustXSendfileTypeHeader()` — задействует передачу запросов на получение файлов веб-серверу (в результате отправку запрошенных файлов выполняет веб-сервер, а не PHP, что повышает производительность).

9.5.2.2. Отправка файла для сохранения на локальном диске

Для отправки файла, который должен быть сохранен на локальном диске, следует вызвать метод `download()` класса `ResponseFactory`:

```
download(<путь к отправляемому файлу>[, <имя файла для сохранения>=null[,
        <ассоциативный массив с добавляемыми в ответ заголовками>=[]]])
```

Если *имя файла для сохранения* не указано, веб-обозреватель сохранит файл под его изначальным именем. Возвращаемый результат также представляется объектом класса `BinaryFileResponse` (см. *разд. 9.5.2.1*). Примеры:

```
return Response::download('c:/sites/common/documents/pricelist.xls');

return Response::download($reportFileName)->deleteFileAfterSend();
```

9.5.2.3. Отправка данных в форматах JSON и JSONP

Для кодирования данных в формат JSON с последующей отправкой клиенту применяется метод `json()` класса `ResponseFactory`:

```
json(<отправляемое значение>[, <код статуса ответа>=200[,
     <ассоциативный массив с добавляемыми в ответ заголовками>=[][,
     <параметры кодирования>=0]]])
```

Параметры кодирования будут переданы функции PHP `json_encode()`, посредством которой и кодируются отправляемые данные. В качестве результата возвращается объект класса `Illuminate\Http\JsonResponse`. Пример:

```
public function rubric(Rubric $rubric) {
    return response()->json($rubric);
}
```

Чтобы преобразовать JSON-данные в формат JSONP, следует у объекта класса `JsonResponse`, возвращенного методом `json()`, вызвать метод `withCallback(<имя функции-обертки>)`. *Имя функции-обертки* может быть произвольным. Пример:

```
return response()->json($rubric)->withCallback('RubricFunction');
```

Также можно воспользоваться методом `jsonp()` класса `ResponseFactory`:

```
jsonp(<имя функции-обертки>, <отправляемое значение>[,
      <код статуса ответа>=200[,
      <ассоциативный массив с добавляемыми в ответ заголовками>=[][,
      <параметры кодирования>=0]]])
```

Пример:

```
return response()->jsonp('RubricFunction', $rubric);
```

Если предназначенное к отправке клиенту значение представляет собой массив, его можно просто вернуть из контроллера:

```
public function platformList() {
    return ['Apache', 'PHP', 'Laravel'];
}
```

В этом случае Laravel самостоятельно закодирует значение в формат JSON перед отправкой.

9.5.2.4. Текстовый ответ

Отправить текстовый ответ просто — достаточно вызвать функцию `response()`:

```
response(<содержание ответа>[, <код статуса ответа>=200[,
        <ассоциативный массив с добавляемыми в ответ заголовками>=[]]])
```

Возвращаемый результат (объект класса `Illuminate\Http\Response`) следует вернуть из контроллера. Пример:

```
public function index() {
    return response('Здесь будет перечень объявлений.', 200,
                    ['Content-Type' => 'text/plain']);
}
```

Если не требуется задавать дополнительные параметры ответа (например, заголовки), можно просто вернуть из контроллера строку с содержанием ответа:

```
public function index() {
    return 'Здесь будет перечень объявлений.';
}
```

Laravel самостоятельно превратит возвращенную строку в полноценный ответ.

9.5.2.5. «Пустой» ответ

«Пустой» ответ создается методом `noContent()` класса `ResponseFactory`:

```
noContent([<код статуса ответа>=200[,
          <ассоциативный массив с добавляемыми в ответ заголовками>=[]]])
```

Пример:

```
return response()->noContent();
```

9.5.3. Дополнительные параметры ответов

Для указания дополнительных параметров ответов (значений их заголовков, кодов статуса и др.) служат приведенные далее методы, поддерживаемые классом `Response` и всеми производными от него, которые были описаны в текущем разделе. Единственное исключение — класс `View`, описанный в *разд. 9.5.1.1* и не являющийся производным от класса `Response`.

В качестве результата все эти методы возвращают объект текущего запроса, что позволяет сцеплять их вызовы.

- `header()` — добавляет в текущий ответ заголовок с заданными *именем* и *значением*:

  ```
  header(<имя заголовка>,
         <значение заголовка>|<массив со значениями заголовка>[,
         <заменить имеющийся заголовок?>=true])
  ```

  Если вторым параметром указать *массив значений*, заголовок получит все эти значения, разделенные запятыми. Если параметру *заменить имеющийся заголовок* дать значение `true`, значение уже присутствующего в текущем ответе заголовка с таким же *именем* будет заменено заданным *значением*. Если же дать этому параметру значение

`false`, заданное *значение* будет добавлено к уже существующему в заголовке через запятую. Пример:

```
public function rubric(Rubric $rubric) {
    return response()->json($rubric)->header('Cache-Control', 'no-cache');
}
```

- `withHeaders()` — добавляет в текущий ответ заголовки из заданного *массива*:

  `withHeaders(<ассоциативный массив с добавляемыми заголовками>)`

  Ключи элементов *ассоциативного массива* должны совпадать с именами заголовков, а их значения задают значения для этих заголовков. Если в ответе уже присутствует какой-либо заголовок из заданных в *массиве*, имеющееся в нем значение будет заменено новым. Пример:

```
public function index() {
    return response('Здесь будет перечень объявлений.')
        ->withHeaders(['Content-Type' => 'text/plain',
                       'Cache-Control' => 'no-cache']);
}
```

- `setStatusCode()` — задает у текущего ответа *статус*:

  `setStatusCode(<код статуса ответа>[, <текстовый статус ответа>=null])`

  Если *текстовый статус ответа* не указан, будет задан типовой статус, соответствующий указанному *коду*. Если в качестве *текстового статуса* задать значение `false`, будет задан статус в виде «пустой» строки. Пример:

```
return response('Объявление не найдено')->setStatusCode(404);
```

- `setCharset(<обозначение кодировки>)` — указывает у текущего ответа текстовую кодировку с заданным *обозначением*.

9.5.4. Перенаправления

Перенаправление можно выполнить на:

- маршрут с заданным *именем* — вызовом функции-хелпера `to_route()`:

```
to_route(<имя маршрута>[,
         <ассоциативный массив со значениями URL-параметров>=[][,
         <код статуса ответа>=302[,
         <ассоциативный массив с добавляемыми в ответ заголовками>=[]]]])
```

  В качестве результата эта и описываемые далее функции возвращают объект класса `Illuminate\Http\RedirectResponse`, представляющий ответ с перенаправлением. Его также следует вернуть из контроллера. Пример:

```
public function store(Request $request) {
    . . .
    return to_route('bbs.detail', ['bb' => $bb->id]);
}
```

  В *ассоциативном массиве URL-параметров* вместо ключа записи можно указать собственно объект, хранящий эту запись, — и фреймворк сам извлечет из него ключ:

```
return to_route('bbs.detail', ['bb' => $bb]);
```

Вместо ключа в качестве значения URL-параметра можно подставить значение любого другого поля или вообще произвольное значение, вычисленное на основе значений любых полей модели. Для этого достаточно переопределить метод соответственно `getRouteKeyName()` или `getRouteKey()` класса модели (подробности — в *разд. 9.4*).

Если в *ассоциативном массиве URL-параметров* указать элементы, не соответствующие записанным в маршруте URL-параметрам, они будут помещены в сгенерированный интернет-адрес в виде GET-параметров:

```
return to_route('bbs.detail', ['bb' => $bb, 'from' => 'userdata']);
// В сгенерированный целевой интернет-адрес будет добавлен
// GET-параметр from со значением userdata
```

- предыдущий интернет-адрес — вызвав функцию `back()`:

```
back([<код статуса ответа>=302[,
     <ассоциативный массив с добавляемыми в ответ заголовками>=[][,
     <запасной интернет-адрес>=false]]])
```

Предыдущий интернет-адрес извлекается из заголовка `Referrer` или, если его в клиентском запросе нет, из серверной сессии. Если адреса предыдущей страницы нет и в сессии, выполняется перенаправление на *запасной интернет-адрес*, а если он не указан — в «корень» сайта. Пример:

```
return back('/home');
```

- произвольный *путь* — вызвав функцию-хелпер `redirect()`:

```
redirect(<целевой путь>[, <код статуса ответа>=302[,
        <ассоциативный массив с добавляемыми в ответ заголовками>=[][,
        <HTTPS?>=null]]])
```

Если параметру *HTTPS* дать значение `true`, перенаправление будет выполнено по протоколу HTTPS, а если дать значение `false` — по протоколу HTTP, если `null` — по текущему протоколу. Пример:

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

Больше инструментов для перенаправления предоставляет класс `Illuminate\Routing\Redirector`, объект которого можно получить, вызвав функцию `redirect()` без параметров или обратившись к фасаду `Illuminate\Support\Facades\Redirect`. Этот класс позволяет выполнить перенаправление на:

- действие контроллера-класса с заданным *именем* — вызвав метод `action()`:

```
action(<контроллер и действие>[,
       <ассоциативный массив со значениями URL-параметров>=[][,
       <код статуса ответа>=302[,
       <ассоциативный массив с добавляемыми заголовками>=[]]]])
```

В качестве первого параметра передается массив из двух строковых элементов — полного пути к контроллеру-классу и имени действия:

```
use App\Http\Controllers\MainController;
. . .
return Redirect::action([MainController::class, 'index']);
```

При вызове этого метода фреймворк просматривает список маршрутов и выполняет перенаправление на адрес, сгенерированный на основе первого маршрута, в котором записаны указанные *контроллер и действие* и который имеет в шаблонном пути заданные в массиве *URL-параметры*;

- произвольный *интернет-адрес* — вызовом метода `away()`:

```
away(<целевой интернет-адрес>[, <код статуса ответа>=302[,
     <ассоциативный массив с добавляемыми в ответ заголовками>=[]]])
```

Пример:

```
return redirect()->away('https://www.google.com/');
```

- маршрут с именем `home` — вызовом метода `home([<код статуса ответа>=302])`;
- текущий интернет-адрес (выполнив тем самым обновление открытой страницы) — вызвав метод `refresh()`:

```
refresh([<код статуса ответа>=302[,
        <ассоциативный массив с добавляемыми заголовками>=[]]])
```

- маршрут с заданным именем — вызовом метода `route()`, аналога функции-хелпера `to_route()`.

В отличие от функции `to_route()`, появившейся только в Laravel 9, метод `route()` поддерживается с самых первых версий фреймворка и часто встречается в старом коде.

Описанные ранее методы в качестве результата также возвращают объект класса `RedirectResponse`.

Класс `RedirectResponse` поддерживает два следующих полезных метода:

- `withFragment(<якорь>)` — добавляет в сгенерированный целевой интернет-адрес указанный *якорь*:

```
return redirect()->route('about')->withFragment('contacts');
// Будет выполнено перенаправление на интернет-адрес
// http://localhost:8000/about#contacts
```

- `withoutFragment()` — удаляет из целевого интернет-адреса присутствующий там якорь.

9.6. Обработка ошибок

При возникновении какой-либо ошибки надо отправить клиенту страницу с соответствующим сообщением. Для этого служат следующие функции-хелперы:

- `abort()` — просто отправляет страницу с сообщением об ошибке с заданным *кодом*:

```
abort(<код ошибки>[, <текстовое сообщение об ошибке>='',
      <ассоциативный массив с добавляемыми в ответ заголовками>=[]])
```

Если *текстовое сообщение* не задано, будет сгенерировано сообщение по умолчанию. Примеры:

```
public function rubric($id) {
    $rubric = Rubric::find($id);
    if ($rubric) {
        return view('rubric', ['rubric' => $rubric]);
    } else {
        abort(404, 'Такая рубрика отсутствует.');
    }
}
```

- `abort_if()` — отправляет сообщение об ошибке с заданным *кодом*, если указанное *условие* истинно:

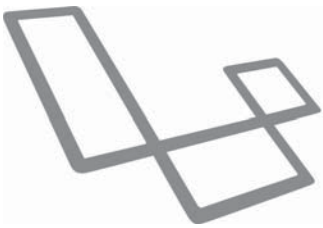
```
abort_if(<условие>, <код ошибки>[, <текстовое сообщение об ошибке>=''[,
        <ассоциативный массив с добавляемыми заголовками>=[]]])
```

Пример:

```
public function rubric($id) {
    $rubric = Rubric::find($id);
    abort_if(($rubric === null), 404, 'Такая рубрика отсутствует.');
    return view('rubric', ['rubric' => $rubric]);
}
```

- `abort_unless()` — отправляет сообщение об ошибке с заданным *кодом*, если указанное *условие* ложно. Формат вызова такой же, как и у функции `abort_if()`. Пример:

```
public function rubric($id) {
    $rubric = Rubric::find($id);
    abort_unless((Auth::check()), 403, 'Сначала выполните вход на сайт');
    return view('rubric', ['rubric' => $rubric]);
}
```



ГЛАВА 10

Обработка введенных данных. Валидация

Данные, введенные посетителем в веб-форму, передаются в составе клиентского запроса. Эти данные следует извлечь и проверить на корректность согласно заданным правилам.

10.1. Извлечение введенных данных

Объект текущего клиентского запроса, содержащий введенные посетителем данные, можно получить способами, описанными в *разд. 9.3*.

Способы получения отдельного значения зависят от типа элемента управления, в который оно было занесено:

- поле ввода (за исключением поля ввода даты и даты и времени), область редактирования, регулятор или скрытое поле — обращением к свойству объекта запроса, чье имя совпадает с наименованием элемента управления (которое задается в атрибуте `name` тега `<input>` или `<textarea>`):

```
<input name="title">
. . .
public function store(Request $request) {
    $bb = new Bb();
    $bb->title = $request->title;
    . . .
}
```

  Также можно воспользоваться методом `input()` объекта запроса:

```
$bb->title = $request->input('title');
```

- флажок — использованием метода `boolean()` объекта запроса. При этом у флажка следует указать в качестве значения (задается атрибутом `value` тега `<input>`) любую из следующих строк: `'1'`, `'true'`, `'on'` или `'yes'`. Пример:

```
<input type="checkbox" name="publish" value="1">
. . .
$bb->publish = $request->boolean('publish');
```

- поле ввода даты или даты и времени — использованием метода `date()` объекта запроса:

```
<input type="date" name="published_at">
. . .
$bb->created_at = $request->date('published_at');
```

- раскрывающийся список с возможностью выбора одного пункта — тем же способом, что используется для извлечения значения из поля ввода:

```
<select name="rubric_id">
    <option value="3">Гаражи</option>
    <option value="9">Дачи</option>
    <option value="2">Дома</option>
    . . .
</select>
. . .
$bb->rubric_id = $request->rubric_id;
```

- раскрывающийся список с возможностью выбора нескольких пунктов — обычно такой элемент управления применяется для связывания записей моделей, между которыми установлена связь «многие-со-многими».

  Наименование списка следует завершить пустыми квадратными скобками (`[]`), сообщив тем самым, что в соответствующем POST-параметре будет сохраняться массив из значений выбранных пунктов, представляющих подлежащие связыванию записи одной модели. Для связывания этих записей с записью другой модели следует применить метод `sync()` (см. *разд. 6.1.4.4*). Пример:

```
<select name="spare_id[]" size="5" multiple>
    <option value="1">Заклепка</option>
    <option value="2">Винт</option>
    <option value="3">Гайка</option>
    . . .
</select>
. . .
Route::patch('/machines/{machine}', 'MachineController@update');
. . .
public function update(Request $request, Machine $machine) {
    . . .
    $machine->spares()->sync($request->input('spare_id'));
    . . .
}
```

Также можно получить все значения, отправленные посетителем, вызвав метод `all()` объекта запроса. Полученный массив можно сразу же передать конструктору класса модели, методам `create()`, `fill()`, `update()` или аналогичным, описанным в *главе 6*. Пример:

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

Если какое-либо значение требует предварительного преобразования в другой формат или тип, его можно преобразовать и занести в запись после вызова упомянутых ранее методов:

```
<input type="number" name="price">
. . .
$rubric = new Rubric($request->all());
if ($request->input('price') == '0')
    $rubric->price = 1;
$rubric->save();
```

«Пустое» значение, указанное у какого-либо элемента управления, по умолчанию преобразуется Laravel в `null`. Это можно использовать для занесения «пустого» значения в необязательное поле таблицы. Далее приведен пример раскрывающегося списка рубрик первого уровня, включающего пункт **– Нет –** с «пустым» значением. Выбор этого пункта вызывает занесение в поле рубрики первого уровня значения `null` — таким образом будет создана рубрика первого уровня:

```
<select name="parent_id">
    <option value="">- Нет -</option>
    <option value="3">Гаражи</option>
    <option value="9">Дачи</option>
    <option value="2">Дома</option>
    . . .
</select>
$rubric = Rubric::create($request->all());
```

10.2. Валидация данных

Валидация — это проверка занесенных посетителем данных на корректность согласно заданным *правилам*, объединенным в *наборы правил*. Каждому элементу управления назначается отдельный набор правил. Валидация всегда выполняется перед использованием этих данных в работе (например, сохранением в базе).

Laravel предоставляет два инструмента для валидации: валидаторы (которые можно создавать как неявно, так и явно) и формальные запросы.

10.2.1. Валидаторы

Валидатор выполняет валидацию данных на уровне отдельного действия контроллера.

10.2.1.1. Быстрая валидация с неявным созданием валидатора

Если нужно просто проверить данные на наличие ошибок ввода, можно выполнить быструю валидацию, при которой валидатор создается неявно, вызвав у объекта текущего запроса метод `validate()`:

```
validate(<наборы правил валидации>[, <сообщения об ошибках>=[][,
        <подстановки наименований>=[]]])
```

Наборы правил валидации указываются в виде ассоциативного массива, ключи элементов которого должны соответствовать именам GET- или POST-параметров (которые, не забываем, совпадают с наименованиями элементов управления, записанными в атрибутах `name` их тегов), содержащих проверяемые данные, а значения элементов зададут для

них сами наборы правил валидации. Правила валидации, поддерживаемые фреймворком, их имена и правила их указания мы рассмотрим позже.

Сообщения об ошибках также указываются в виде ассоциативного массива, ключи элементов которого должны быть записаны в формате:

<имя GET- или POST-параметра>.<имя правила валидации>

Значения элементов этого массива зададут сами текстовые сообщения об ошибках. Правила их написания мы рассмотрим позже.

Подстановки наименований, задаваемые в последнем параметре метода `validate()`, мы также рассмотрим позже.

> ***Если сообщения об ошибках не указаны...***
>
> ...будут выводиться сообщения по умолчанию. Если не была выполнена локализация сайта (описываемая в *главе 28*), сообщения будут на английском языке.

Если валидация прошла успешно, в качестве результата метод возвращает ассоциативный массив с проверяемыми данными, подготовленными к обработке. Этот массив может быть передан конструктору класса модели, методам `create()`, `fill()`, `update()` и др.

В случае неуспеха валидации:

- ❒ введенные данные будут сохранены в серверной сессии (с целью занести их в веб-форму при повторном ее выводе, чтобы посетитель смог внести исправления);
- ❒ список сообщений об ошибках ввода также будет сохранен в серверной сессии — в особом *хранилище ошибок* (с той же целью);
- ❒ на экран будет автоматически выведена предыдущая страница (практически всегда это страница, содержащая веб-форму, в которую посетитель ввел данные) с кодом статуса 422 (введенные данные содержат ошибку).

Пример:

```
public function store(Request $request) {
    $validationRules = ['title' => 'required|max:50',
                        'price' => 'required|numeric'];

    $errorMessages = [
        'title.required' => 'Введите название товара',
        'title.max' => 'Название не может быть длиннее 50 символов',
        'title.required' => 'Введите цену товара',
        'title.numeric' => 'Цена должна представлять собой число',
    ];
    $validated = $request->validate($validationRules, $errorMessages);
    $request->user()->bbs()->create($validated);
    . . .
}
```

Метод `validate()` сохраняет список сообщений о допущенных ошибках в хранилище ошибок по умолчанию. Однако можно поместить их в *именованное хранилище* с произвольным именем (что может быть полезно, если на странице находятся несколько веб-форм). Для этого достаточно использовать метод `validateWithBag()`:

```
validateWithBag(<имя хранилища ошибок>, <наборы правил валидации>[,
                <сообщения об ошибках>=[][, <подстановка наименований>=[]]])
```

Пример:

```
$validated = $request->validateWithBag('bb', $validationRules, $errorMessages);
```

Поддержка методов `validate()` и `validateWithBag()` реализована в трейте `Illuminate\Foundation\Validation\ValidatesRequests`, который изначально включается в базовый контроллер `App\Http\Controllers\Controller`.

10.2.1.2. Валидация с явным созданием валидатора

Явное создание валидатора предоставляет ряд дополнительных возможностей, которые могут оказаться полезными. Оно выполняется вызовом метода `make()` у фасада `Illuminate\Support\Facades\Validator`:

```
make(<ассоциативный массив с проверяемыми данными>, <наборы правил валидации>[,
     <сообщения об ошибках>=[][, <подстановки наименований>=[]]])
```

Метод возвращает объект класса `Illuminate\Validation\Validator`, представляющий созданный валидатор. Пример:

```
use Illuminate\Support\Facades\Validator;
. . .
$validator = Validator::make($request->all(), $validationRules,
                                              $errorMessages);
```

Вместо метода `make()`, можно использовать функцию-хелпер `validator()`, имеющую тот же формат вызова, — это немного сократит код:

```
$validator = validator($request->all(), $validationRules, $errorMessages);
```

Однако функция `validator()` исполняется чуть медленнее метода `make()`.

Метод `sometimes()` класса `Validator` позволяет добавить в текущий валидатор указанный набор правил валидации, если выполняется заданное условие:

```
sometimes(<имя GET- или POST-параметра>, <набор правил валидации>,
                                         <анонимная функция>)
```

Заданный *набор правил валидации* будет назначен параметру с указанным *именем*, если указанная *анонимная функция* вернет `true`. Эта *функция* в качестве параметра принимает объект, хранящий все GET- и POST-параметры, полученные с текущим запросом. Пример:

```
$validator->sometimes('hobbies', 'required|string',
                      function ($input) {
                          return $input->you_have_hobby == true;
                      });
```

Для выполнения собственно валидации класс `Validator` предлагает следующие высокоуровневые методы:

- `validate()` — аналогичен одноименному методу объекта запроса (см. *разд. 10.2.1.1*):

```
$validated = $validator->validate();
$rubric = new Rubric($validated);
```

- `validateWithBag(<имя хранилища ошибок>)` — аналогичен одноименному методу объекта запроса (см. *разд. 10.2.1.1*).

Далее приведены низкоуровневые методы этого класса:

- `passes()` — выполняет валидацию и возвращает `true`, если данные прошли валидацию, и `false` — в противном случае;
- `validated()` — возвращает ассоциативный массив данных, прошедших валидацию.

  Пример:

```
if ($validator->passes()) {
    // Если валидация прошла успешно, сохраняем новую рубрику и выполняем
    // перенаправление на список рубрик
    $bb = new Bb($validator->validated());
    $bb->save();
    return redirect()->route('bb.index');
} else {
    // В противном случае идем на страницу добавления рубрики
    return redirect()->route('bb.create');
}
```

- `fails()` — выполняет валидацию и возвращает `true`, если данные *не* прошли валидацию, и `false` — в противном случае:

```
if ($validator->fails())
    return redirect()->route('rubric.create');
```

  Если метод `fails()` в процессе валидации находит какое-либо значение некорректным, он *не* останавливает дальнейшую проверку;

- `stopOnFirstFailure()` — указывает валидатору прервать валидацию при нахождении первого некорректного значения. Вызывается перед вызовом метода `fails()`. Пример:

```
if ($validator->stopOnFirstFailure()->fails())
    return redirect()->route('rubric.create');
```

- `after(<анонимная функция>)` — задает *анонимную функцию*, которая будет вызвана после валидации и может выполнить какие-либо дополнительные проверки. В качестве единственного параметра она должна принимать текущий объект валидатора и не должна возвращать результат.

  Для получения хранилища ошибок в теле *анонимной функции* применяется метод `errors()` валидатора. Для добавления нового сообщения об ошибке служит метод `add()` хранилища ошибок:

```
add(<имя GET- или POST-параметра>, <сообщение об ошибке>)
```

  Пример:

```
$validator->after(function ($validator) {
    if ($validator->validated()['title'] == 'Вечные ценности') {
        $validator->errors()->add('name', 'Не продаются!');
    }
});
```

- `safe([<массив наименований>=null])` — возвращает объект класса `Illuminate\Support\ValidatedInput`, хранящий данные, которые прошли валидацию.

  Объект этого класса поддерживает функциональность ассоциативного массива…

  ```
  $vi = $validator->safe();
  $title = $vi['title'];
  ```

  …равно как и функциональность коллекции, вследствие чего его можно перебрать в цикле:

  ```
  foreach ($vi as $key => $value) {
      . . .
  }
  ```

  Однако его нельзя использовать вместо настоящего ассоциативного массива (например, передать конструктору или методу `fill()` модели).

  Класс `ValidatedInput` поддерживает следующие методы:

  - `all()` — возвращает ассоциативный массив со всеми значениями, хранящимися в текущем объекте:

    ```
    $bb->fill($vi->all());
    ```

  - `only()` — возвращает ассоциативный массив, содержащий только значения с указанными *наименованиями*. Форматы вызова:

    ```
    only(<наименование 1>, <наименование 2> ... <наименование n>)
    only(<массив с наименованиями>)
    ```

    Пример:

    ```
    $bb->fill($vi->only('title', 'content', 'address', 'price'));
    ```

  - `except()` — возвращает ассоциативный массив, содержащий только значения, наименования которых, напротив, *не* были указаны. Форматы вызова такие же, как у метода `only()`;

  - `collect()` — возвращает коллекцию `Collection`, со всеми значениями, хранящимися в текущем объекте;

  - `merge(<массив со значениями>)` — добавляет в текущий объект значения из указанного ассоциативного *массива*. Ключи этого *массива* зададут наименования добавляемых значений, а значения элементов — собственно добавляемые значения. Пример:

    ```
    $vi->merge(['user_id' => Auth::user()->id]);
    $bb = new Bb($vi->all());
    ```

При использовании низкоуровневых методов в случае неуспеха валидации введенные данные и сообщения об ошибках *не* сохраняются в серверной сессии автоматически. Это придется сделать вручную при перенаправлении, вызывая приведенные далее методы, поддерживаемые объектом ответа с перенаправлением (см. *разд. 9.5.4*):

- `withInput()` — сохраняет в серверной сессии все GET- и POST-параметры, полученные в запросе;

- `onlyInput()` — сохраняет в серверной сессии только GET- и POST-параметры с указанными в вызове *именами*:

  `onlyInput(<имя параметра 1>, <имя параметра 2> ... <имя параметра n>)`

- `exceptInput()` — сохраняет в серверной сессии все GET- и POST-параметры, кроме имеющих указанные в вызове *имена*. Формат вызова такой же, как и у метода `onlyInput()`.

  Можно указать GET- и POST-параметры, значения которых никогда не будут сохраняться в сессиях (обычно так поступают в случае параметров, хранящих крайне конфиденциальные данные). Для этого нужно открыть модуль с классом стандартного обработчика исключений `App\Exceptions\Handler` и добавить имена нужных параметров в массив, присвоенный защищенному свойству `dontFlash` этого класса. Пример:

```
class Handler extends ExceptionHandler {
    . . .
    protected $dontFlash = [
        . . .
        'secret_phrase',
    ];
    . . .
}
```

Изначально в этом массиве присутствуют параметры `current_password`, `password` и `password_confirmation`;

- `withErrors()` — сохраняет в серверной сессии все сообщения об ошибках ввода:

  `withErrors(<объект валидатора>[, <имя хранилища ошибок>='default'])`

  Если *имя хранилища ошибок* не указано, ошибки будут помещены в хранилище по умолчанию.

Все эти методы в качестве результата возвращают текущий объект перенаправления, что позволяет сцеплять их вызовы. Пример:

```
if ($validator->fails())
    return redirect()->route('bb.create')->withInput()->withErrors($validator);
```

10.2.2. Формальные запросы

Формальный запрос — это клиентский запрос с расширенной функциональностью, самостоятельно выполняющий валидацию поступивших данных. Применение формальных запросов для валидации данных позволяет несколько сократить код контроллеров.

Новый формальный запрос создается командой:

`php artisan make:request <имя формального запроса>`

Классы формальных запросов объявляются в пространстве имен `App/Http/Requests` (соответствующая папка будет создана автоматически) и являются производными от класса `Illuminate\Foundation\Http\FormRequest`, в свою очередь, производного от класса запроса `Request`.

В новом классе формальных запросов следует объявить следующие общедоступные методы:

- `rules()` — должен возвращать ассоциативный массив с наборами правил валидации, аналогичный указываемому в вызове метода `validate()` (см. *разд. 10.2.1.1*). Изначально возвращает «пустой» массив;
- `messages()` — должен возвращать ассоциативный массив с сообщениями об ошибках, также аналогичный указываемому в вызове метода `validate()`;
- `authorize()` — должен возвращать `true`, если текущий пользователь имеет привилегии на выполнение операции (например, правку или удаление объявления, привилегии на которое имеет лишь пользователь, оставивший это объявление), и `false` — если он не имеет привилегии (так, пользователь не имеет привилегии на удаление чужого объявления). Изначально возвращает значение `false` (тем самым запрещая любые операции).

  Более подробно переопределение этого метода будет рассмотрено в *главе 13*, рассказывающей о разграничении доступа. Сейчас же следует исправить этот метод, чтобы он всегда возвращал значение `true`;
- `withValidator($validator)` — в единственном параметре `validator` принимает объект создаваемого неявно валидатора и может задать у него какую-либо дополнительную логику вызовом метода `after()` (см. *разд. 10.2.1.2*);
- `attributes()` — служит для подстановки наименований, которая будет описана позже.

Переопределив в формальном запросе защищенный метод `prepareForValidation()`, можно выполнить какую-либо предварительную обработку данных перед валидацией. Извлечь исходные данные из текущего запроса можно способами, описанными в *разд. 9.3*. Чтобы занести в запрос результаты преобразования этих данных, нужно использовать следующие методы, поддерживаемые классом запроса:

- `merge(<`*массив значений*`>)` — добавляет в состав данных, присутствующих в текущем запросе, значения, приведенные в заданном ассоциативном *массиве*. Ключи элементов этого *массива* зададут имена у добавляемых значений. Если в текущем запросе уже существует значение с заданным именем, оно будет перезаписано. Пример:

```
use Illuminate\Support\Str;
. . .
// Добавляем в запрос значение slug, хранящее слаг рубрики
$request->merge(['slug' => Str::slug($request->name)]);
```

- `replace(<`*массив значений*`>)` — полностью заменяет все значения, присутствующие в текущем запросе, значениями, приведенными в заданном ассоциативном *массиве*.

По умолчанию формальный запрос *не* прерывает валидацию при нахождении первого некорректного значения и, если данные не прошли валидацию, производит перенаправление на предыдущую страницу с кодом статуса 422 (получены ошибочные данные). Однако это поведение можно изменить, переопределив в классе формального запроса следующие защищенные свойства:

- `stopOnFirstFailure` — если `true`, валидация будет прервана при нахождении первого некорректного значения, если `false` — не будет прервана (по умолчанию: `false`);

- `redirect` — строка с путем, по которому будет выполнено перенаправление в случае некорректности полученных данных (по умолчанию: `null`);
- `redirectRoute` — строка с именем маршрута, по которому будет выполнено перенаправление в случае некорректности полученных данных (по умолчанию: `null`).

В листинге 10.1 показан пример класса формального запроса, обрабатывающего запрос с создаваемой рубрикой.

Листинг 10.1. Пример класса формального запроса, обрабатывающего создаваемую рубрику

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Support\Str;

class RubricCreateRequest extends FormRequest {
    protected $stopOnFirstFailure = true;

    public function rules() {
        return [
            'name' => 'required|max:40|unique:rubrics',
            'parent_id' => 'exists:rubrics,id|nullable',
        ];
    }

    public function messages() {
        return [
            'name.required' => 'Введите название рубрики',
            'name.max' => 'Название не может быть длиннее :max символов',
            'name.unique' => 'Такая рубрика уже существует',
            'parent_id.exists' =>
                            'Указана несуществующая рубрика первого уровня'
        ];
    }

    public function authorize() {
        return true;
    }

    public function withValidator($validator) {
        $validator->after(function ($validator) {
            if ($validator->validated()['name'] == 'Вечные ценности') {
                $validator->errors()->add('name', 'Не продаются!');
            }
        });
    }

    protected function prepareForValidation() {
        $this->merge(['name' => Str::ucfirst($this->name)]);
    }
}
```

Чтобы выполнить валидацию полученных данных с помощью формального запроса, следует у параметра действия контроллера, в который подставляется объект клиентского запроса, указать в качестве типа нужный класс формального запроса. Пример:

```
use App\Http\Requests\RubricRequest;
. . .
public function store(RubricRequest $request) {
    Rubric::create($request->all());
    . . .
}
```

10.2.3. Правила валидации и написание их наборов

Как говорилось в *разд. 10.2.1.1*, совокупность наборов правил валидации для проверки данных, полученных в составе запроса, записывается в виде ассоциативного массива. Ключи элементов массива должны совпадать с именами GET- и POST-параметров, присутствующих в запросе, а значения элементов зададут собственно наборы правил валидации для этих параметров.

Отдельный набор правил валидации может содержать произвольное количество отдельных правил и записывается в одном из следующих форматов:

- в виде строки — отдельные правила разделяются символами вертикальной черты (`|`):

  ```
  $validationRules = ['title' => 'required|max:50'];
  ```

- в виде массива — отдельные правила оформляются в виде его элементов:

  ```
  $validationRules = ['title' => ['required', 'max:50']];
  ```

  Этот формат применяется в случаях, если какие-либо правила из набора задаются не в виде строк, а в виде объектов соответствующих классов (будут описаны далее).

Многие правила валидации принимают параметры. Значения параметров записываются через двоеточие (`:`) после имени правила валидации и отделяются друг от друга запятыми. Пример: `digits_between:2,50`.

Далее приведены правила валидации, поддерживаемые Laravel, и значения GET- и POST-параметров, которые должны удовлетворять этим правилам:

- `required` — любое «непустое» значение (т. е. не являющееся `null`, «пустой» строкой или «пустым» массивом). Текущий параметр обязан присутствовать в запросе. Пример:

  ```
  'title' => 'required|string|max:50'
  ```

- `present` — любое «пустое» или «непустое» значение. Текущий параметр обязан присутствовать в запросе. Обычно используется с необязательными для заполнения элементами управления;

  ```
  'addendum' => 'present'
  ```

- `filled` — любое «непустое» значение, но только если текущий параметр присутствует в запросе. Обычно используется в случаях, когда соответствующий элемент

управления может быть недоступным для ввода (например, если сброшен какой-либо флажок) или вообще отсутствовать в веб-форме. Пример:

```
'hobbies' => 'filled|string'
```

- `sometimes` — любое «пустое» или «непустое» значение, при этом текущий параметр может отсутствовать в запросе. Применяется вместе с другими правилами. Пример:

  ```
  'hobbies' => 'sometimes|required|string'
  ```

- `string` — строка, содержащая любые символы;
- `alpha` — строка, содержащая только буквы;
- `alpha_num` — строка, содержащая только буквы и цифры;
- `alpha_dash` — строка, содержащая только буквы, дефисы и символы подчеркивания;
- `numeric` — целое или вещественное число;
- `integer` — целое число;
- `date` — значение даты в формате, поддерживаемом функцией PHP `strtotime()`;
- `timezone` — обозначение временно́й зоны из числа возвращаемых функцией PHP `timezone_identifiers_list()`;
- `boolean` — логическая величина, представленная в виде строки: `'true'` и `'1'`, обозначает логическую истину (`true`), `'false'` и `'0'` — ложь (`false`). Применяется совместно с флажками;
- `url` — произвольный интернет-адрес;
- `active_url` — только реально существующий интернет-адрес (его существование проверяется);
- `email[:<`*валидаторы через запятую*`>]` — адрес электронной почты. В параметре можно указать следующие *валидаторы* электронной почты, используемые для проверки адреса:
  - `rfc` — обычная проверка соответствия адреса стандартам (его существование в реальности не проверяется);
  - `strict` — аналогично `rfc`, но проверка более строгая;
  - `dns` — проверяет, существует ли почтовый сервер, присутствующий в заданном адресе, на самом деле;
  - `spoof` — проверяет, не присутствуют ли в адресе недопустимые символы;
  - `filter` — использует средства валидации адресов, встроенные в PHP.

  Пример:

  ```
  'user_email' => 'email:rfc,dns,spoof'
  ```

- `ip` — IP-адрес любого формата;
- `ipv4` — IP-адрес формата IPv4;
- `ipv6` — IP-адрес формата IPv6;
- `mac_address` — MAC-адрес;

- `uuid` — универсальный уникальный идентификатор;
- `json` — данные в формате JSON;
- `array[:<ключи через запятую>]` — массив, индексированный или ассоциативный. Можно указать *ключи*, элементы с которыми обязательно должны присутствовать в массиве, при этом элементы с неуказанными ключами в процессе валидации будут удалены. Пример:

  ```
  'bb_data' => 'array:title,content,address,price,rubric_id,publish'
  ```

- `required_array_keys:<ключи через запятую>` — то же, что и `array`, только элементы с неуказанными ключами *не* удаляются из массива;
- `nullable` — допускается значение `null`:

  ```
  'parent_id' => 'integer|nullable'
  ```

- `current_password[:<страж>]` — значение, совпадающее с паролем текущего пользователя:

  ```
  'retype_your_password_to_continue' => 'current_password'
  ```

  Можно указать имя используемого *стража* (если не указан, будет использован страж по умолчанию, — подробно о стражах будет рассказано в *главе 13*):

  ```
  'retype_your_password_to_continue' => 'current_password:api'
  ```

- `password` — то же самое, что и `current_password`. Это правило сохранено для совместимости со старыми версиями Laravel и в следующих версиях будет удалено;
- `confirmed` — значение, совпадающее со значением из GET- или POST-параметра с именем формата `<имя текущего параметра>_confirmation` (например, значение параметра `password` должно совпадать со значением параметра `password_confirmation`):

  ```
  'password' => 'confirmed'
  ```

- `accepted` — одна из строк: `'yes'`, `'on'`, `'1'` или `'true'`. Обычно применяется с флажками вида **Я прочитал(а) пользовательское соглашение**;
- `declined` — одна из строк: `'no'`, `'off'`, `'0'` или `'false'`;
- `required_if:<other>,<values>` — любое «непустое» значение, если значение из параметра с именем *other* равно какой-либо величине из списка *value* (в котором отдельные величины разделяются запятыми):

  ```
  'hobbies_list' => 'required_if:has_hobby,true,1,yes'
  ```

  Это правило валидации можно создать программно, вызвав у фасада `Illuminate\Validation\Rule` метод `requiredIf()`:

  ```
  requiredIf(<условие>|<анонимная функция>)
  ```

  Можно указать *условие*, выдающее в качестве результата логическую величину: `true` требует обязательного ввода текущего значения, `false` делает его необязательным для заполнения:

  ```
  use Illuminate\Validation\Rule;
  . . .
  'user_role' => [Rule::requiredIf(request()->user()->bbs()->count() > 0)]
  ```

Также можно указать *анонимную функцию*, которая не принимает параметров и возвращает также логическую величину:

```
'user_role' => [Rule::requiredIf(function () {
                        return request()->user()->bbs()->count() > 0;
                    })]
```

- `required_unless:<other>,<values>` — любое «непустое» значение, если значение из параметра с именем *other* **не** равно ни одной величине из списка *values* (отдельные величины разделяются запятыми):

  ```
  'hobbies_list' => 'required_unless:has_hobby,false,0,no'
  ```

- `required_with:<values>` — любое «непустое» значение, если в запросе присутствует и не является «пустым» какой-либо из параметров, имена которых приведены в списке *values* через запятую;

- `required_with_all:<values>` — любое «непустое» значение, если в запросе присутствуют и не являются «пустыми» все параметры, имена которых приведены в списке *values* через запятую;

- `required_without:<values>` — любое «непустое» значение, если в запросе отсутствует или является «пустым» какой-либо из параметров, имена которых приведены в списке *values* через запятую;

- `required_without_all:<values>` — любое «непустое» значение, если в запросе отсутствуют или являются «пустыми» все параметры, имена которых приведены в списке *values* через запятую;

- `accepted_if:<other>,<value>` — одна из строк: `'yes'`, `'on'`, `'1'` или `'true'`, если значение из параметра с именем *other* равно величине *value*:

  ```
  'mail_agreed' => 'accepted_if:send_me_offers,true'
  ```

- `declined_if:<other>,<value>` — одна из строк: `'no'`, `'off'`, `'0'` или `'false'`, если значение из параметра *other* равно величине *value*;

- `same:<other>` — значение, совпадающее со значением из параметра с именем *other*;

  ```
  'password_confirm' => 'same:password'
  ```

- `different:<other>` — значение, отличающееся от значения из параметра с именем *other*;

- `starts_with:<values>` — строка, которая начинается на одну из подстрок, приведенных в списке *values* через запятую:

  ```
  'php_variable_name' => 'starts_with:$'
  ```

- `ends_with:<values>` — строка, которая оканчивается на одну из подстрок, приведенных в списке *values* через запятую:

  ```
  'executable_file_name' => 'ends_with:com,exe,cmd,bat'
  ```

- `in:<величины через запятую>` — значение, совпадающее с одной из указанных *величин*:

  ```
  'user_role' => 'required|in:admin,editor,author'
  ```

Сформировать правило можно программно, вызвав у фасада `Rule` метод `in()`, поддерживающий два формата вызова:

```
in(<величина 1>, <величина 2> ... <величина n>)
in(<массив с величинами>)
```

Пример:

```
use Illuminate\Validation\Rule;
. . .
'user_role' => [Rule::in(['admin', 'editor', 'author'])]
```

- `not_in:<величины через запятую>` — значение, *не* совпадающее ни с одной из указанных *величин*. Для программного формирования этого правила применяется метод `notIn()` фасада `Rule`. В остальном аналогично правилу `in`;
- `digits:<digits>` — число, содержащее строго *digits* цифр:

  ```
  'post_index" => 'digits:6'
  ```

- `digits_between:<min>,<max>` — число, содержащее от *min* до *max* цифр;
- `multiple_of:<value>` — значение, кратное величине *value*;
- `date_format:<format>` — значение даты в формате, заданном величиной *format*. Можно указать любой формат даты, поддерживаемый классом PHP `DateTime`.

  > ***СЛЕДУЕТ ИСПОЛЬЗОВАТЬ ЛИБО ПРАВИЛО DATE, ЛИБО DATE_FORMAT***
  > Но не оба одновременно!

  Пример:

  ```
  'published_at' => 'date_format:d.m.Y'
  ```

- `date_equals:<date>` — значение даты, равное величине *date*. В качестве последней можно задать либо строку в формате, поддерживаемом функцией PHP `strtodate()`, либо имя какого-либо параметра — тогда величина даты для сравнения будет взята из него. Примеры:

  ```
  'new_year_at' => 'date|date_equals:2020/01/01',
  'signed_at' => 'date|date_equals:given_at'
  ```

- `before:<date>` — значение даты, меньшее величины *date* (поддерживаются те же форматы, что приведены в описании правила `date_equals`):

  ```
  'published_at' => 'date|before:actual_to'
  ```

- `before_or_equal:<date>` — значение даты, меньшее или равное величине *date* (см. описание правила `date_equals`);
- `after:<date>` — значение даты, большее величины *date* (см. описание правила `date_equals`):

  ```
  'published_at' => 'date|after:created_at'
  ```

- `after_or_equal:<date>` — значение даты, большее или равное величине *date* (см. описание правила `date_equals`);

- `size:<value>` — значение, равное величине *value*. Допускается сравнивать числа, строки (у них сравнивается длина) и массивы (у них — размер). Пример:

  ```
  'post_code' => 'string|size:6'
  ```

- `min:<min>` — значение, большее или равное величине *min* (особенности сравнения данных разных форматов приведены в описании правила `size`):

  ```
  'year' => 'integer|min:1970'
  ```

- `max:<max>` — значение, меньшее или равное величине *min* (см. описание правила `size`);

- `between:<min>,<max>` — значение, находящееся в диапазоне от *min* до *max* включительно (см. описание правила `size`);

- `lt:<value>` — значение, меньшее значения из параметра с именем *value* (см. описание правила `size`):

  ```
  'start_price' => 'numeric|lt:end_price'
  ```

- `lte:<value>` — значение, меньшее или равное значению из параметра с именем *value* (см. описание правила `size`);

- `gt:<value>` — значение, большее значения из параметра с именем *value* (см. описание правила `size`);

- `gte:<value>` — значение, большее или равное значению из параметра с именем *value* (см. описание правила `size`);

- `regex:<регулярное выражение>` — значение, совпадающее с заданным *регулярным выражением*. Последнее указывается в формате, поддерживаемом функцией PHP `preg_match()`;

- `not_regex:<регулярное выражение>` — значение, *не* совпадающее с заданным *регулярным выражением*. Последнее указывается в формате, поддерживаемом функцией PHP `preg_match()`;

- `exists:<таблица>[,<поле>]` — значение, хранящееся в заданном *поле* указанной *таблицы*. Если *поле* не указано, поиск значения будет выполняться в поле, чье имя совпадает с именем текущего GET- или POST-параметра.

  В параметре *таблица* можно указать:

  - имя таблицы — если она хранится в базе данных, используемой по умолчанию;
  - строку формата *<имя базы данных>.<имя таблицы>* — если требуемая таблица хранится в базе данных, отличной от используемой по умолчанию;
  - полный путь к классу модели, обслуживающей эту таблицу.

  Примеры:

  ```
  'username' => 'exists:users'

  'username' => 'exists:App\\Models\\User'

  'author_name' => 'exists:my_users,username'
  ```

Сформировать это правило можно программно, вызвав у фасада `Rule` метод `exists()`:

```
exists(<имя таблицы>[, <имя поля>=null])
```

Пример:

```
use Illuminate\Validation\Rule;
. . .
'author_name' => [Rule::exists('sqlite.users', 'username')]
```

По умолчанию будут просматриваться все записи заданной *таблицы*. Чтобы просматривались только записи, удовлетворяющие заданным условиям фильтрации, следует у объекта правила, возвращенного методом `exists()`, вызвать следующие методы:

- `where()` — отбирает записи, хранящие в поле с заданным *именем* указанное *значение*. Поддерживаются два формата вызова:

  ```
  where(<имя поля>[, <значение>=null])
  where(<анонимная функция>)
  ```

  Пример:

  ```
  'author_name' => [Rule::exists('users', 'username')
                       ->where('active', true)]
  ```

  Если в качестве *значения* задать массив, будут отбираться записи, содержащие в этом поле любое из значений, которые присутствуют в заданном массиве.

  Вызовы этого метода можно сцеплять друг с другом — тогда задаваемые ими условия фильтрации будут объединяться по правилам логического И:

  ```
  'author_name' => [Rule::exists('users', 'username')
                     ->where('active', true)->where('blocked', false)]
  ```

  Второй формат вызова позволяет создать более сложное условие фильтрации. Задаваемая *анонимная функция* в качестве параметра должна принимать объект построителя запросов и, вызывая его методы (были описаны в *разд. 7.3.4*), создавать нужное условие фильтрации. Пример:

  ```
  'author_name' => [Rule::exists('users', 'username')
                       ->where(function ($query) {
                              $query->where('role', 'admin')
                                    ->orWhere('role', 'editor');
                         })]
  ```

- `whereNot(<имя поля>, <значение>)` — отбирает записи, *не* содержащие в поле с заданным *именем* указанное *значение*;
- `whereNull(<имя поля>)` — отбирает записи, хранящие в поле с заданным *именем* значение `null`;
- `whereNotNull(<имя поля>)` — отбирает записи, хранящие в поле с заданным *именем* значение, отличное от `null`;
- `whereIn(<имя поля>, <массив>)` — отбирает записи, хранящие в поле с указанным *именем* одно из значений, которые содержатся в заданном *массиве*;

- whereNotIn(<*имя поля*>, <*массив*>) — отбирает записи, *не* хранящие в поле с указанным *именем* ни одного из значений, которые содержатся в заданном *массиве*;
- withoutTrashed() — исключает из результата записи, подвергшиеся «мягкому» удалению:

  withoutTrashed([<*имя поля отметки удаления*>='deleted_at'])

❐ unique:<*таблица*>[,<*поле*>] — значение, отсутствующее в заданном *поле* указанной *таблицы*. Значения обоих параметров указываются в таком же формате, что и у правила exists. Пример:

```
'name' => 'required|unique:rubrics'
```

Это правило можно сформировать программно, вызвав у фасада Rule метод unique(), имеющий тот же формат вызова, что и метод exists():

```
use Illuminate\Validation\Rule;
. . .
'name' => [Rule::unique('rubrics')]
```

Объект правила, возвращаемый методом unique(), поддерживает все методы объекта, возвращаемого методом exists():

```
// Исключаем из рассмотрения запись, хранящую пользователя, сведения
// о котором исправляются в настоящий момент
'username' => [Rule::unique('users')
                   ->whereNot('id', $current_user->id)]
```

Еще он поддерживает метод ignore(), который исключает из рассмотрения заданный объект модели:

ignore(<*исключаемая запись*>[, <*имя ключевого поля*>=null])

В качестве *исключаемой записи* можно указать:

- ключ этой записи. Если ключевое поле обслуживаемой таблицы имеет имя, отличное от id, его следует указать во втором параметре;
- объект модели, хранящий эту запись.

Пример:

```
'username' => [Rule::unique('users')->ignore($current_user)]
```

❐ distinct[:[strict][,][ignore_case]] — массив, содержащий лишь уникальные, неповторяющиеся значения:

```
'spares.*.name' => 'array|distinct'
```

По умолчанию фреймворк в процессе поиска дубликатов в массиве выполняет нестрогое сравнение значений с учетом регистра символов. Если указан параметр strict, будет выполняться строгое сравнение, если указан параметр ignore_case — без учета регистра. Пример:

```
'spares.*.name' => 'array|distinct:strict,ignore_case'
```

❐ in_array:<*other*> — значение, содержащееся в массиве из параметра с именем *other*:

```
'main_title' => 'in_array:titles.*'
```

- `exclude` — текущее значение будет исключено из массива проверенных данных, возвращаемого методами `validate()` и `validated()`:

  ```
  'unneeded_value' => 'exclude'
  ```

- `exclude_if:<other>,<value>` — текущее значение будет исключено из массива проверенных данных, возвращаемого методами `validate()` и `validated()`, если значение из параметра с именем *`other`* равно величине *`value`*:

  ```
  'unneeded_value' => 'exclude_if:exclude_unneeded_values,true'
  ```

  Если требуется реализовать более сложную логику исключения текущего значения из массива проверенных данных, можно воспользоваться методом `excludeIf()`, вызываемым у фасада `Rule`:

  ```
  excludeIf(<исключать текущее значение?>|<анонимная функция>)
  ```

  Параметру *`исключать текущее значение`* можно дать значение `true` (исключить текущее значение из проверенных данных) или `false` (не исключать). Также можно задать *`анонимную функцию`*, не принимающую параметров и возвращающую либо `true`, либо `false`. Пример:

  ```
  'unneeded_value' => [Rule::excludeIf(fn () =>
                       $request->boolean('exclude_unneeded_values') === true]
  ```

- `exclude_unless:<other>,<value>` — текущее значение будет исключено из массива проверенных данных, возвращаемого методами `validate()` и `validated()`, если значение из параметра с именем *`other`* ***не*** равно величине *`value`*.

  Если в качестве величины *`value`* указано значение `null`, текущее значение будет исключено, если значение из параметра *`other`* не равно `null` или если этот параметр отсутствует в запросе;

- `exclude_with:<other>` — текущее значение будет исключено из массива проверенных данных, возвращаемого методами `validate()` и `validated()`, если параметр с именем *`other`* присутствует в запросе;

- `exclude_unless:<other>` — текущее значение будет исключено из массива проверенных данных, возвращаемого методами `validate()` и `validated()`, если параметр с именем *`other`* отсутствует в запросе;

- `prohibited` — текущий GET- или POST-параметр должен либо быть «пустым», либо вообще отсутствовать в запросе;

- `prohibited_if:<other>,<value>` — текущий параметр должен либо быть «пустым», либо отсутствовать в запросе, если значение из параметра с именем *`other`* равно величине *`value`*. Для программного формирования правила можно использовать метод `prohibitedIf()` фасада `Rule`. В остальном аналогично правилу `exclude_if`;

- `prohibited_unless:<other>,<value>` — текущий параметр должен либо быть «пустым», либо отсутствовать в запросе, если значение из параметра с именем *`other`* ***не*** равно величине *`value`*;

- `prohibits:<имена параметров через запятую>` — если текущий параметр присутствует в запросе, параметры с заданными *`именами`* должны в нем отсутствовать, даже если они «пусты»;

- `bail` — остановить валидацию, если значение из текущего параметра не удовлетворяет одному из правил, указанных в наборе:

```
'price' => 'required|numeric|bail'
```

Можно проверять текущее значение на равенство какому-либо варианту заданного типизированного перечисления. Для этого используется валидатор, представляемый классом `Illuminate\Validation\Rules\Enum`. В качестве правила указывается объект этого класса. Конструктор класса `Enum` имеет формат: `Enum(<`*полное имя перечисления*`>)`. Пример:

```
enum BbType: int {
    case BUY = 1;
    case SELL = 2;
}
. . .
use Illuminate\Validation\Rules\Enum;
. . .
'bb_kind' => [new Enum(BbType::class)]
```

10.2.3.1. Валидация паролей

Для выполнения специфической валидации пользовательских паролей служит класс `Illuminate\Validation\Rules\Password`. Вот методы этого класса, задающие отдельные правила для валидации паролей, и пароли, удовлетворяющие этим правилам:

- `min(<`*длина*`>)` — пароль, имеющий длину не менее указанной (задается в символах);
- `letters()` — пароль, содержащий как минимум одну букву в любом регистре;
- `mixedCase()` — пароль, содержащий как минимум одну букву в верхнем регистре и одну букву в нижнем регистре;
- `numbers()` — пароль, содержащий как минимум одну цифру;
- `symbols()` — пароль, содержащий как минимум один пробел, знак пунктуации или иной символ, который не является буквой или цифрой;
- `uncompromised([<`*количество утечек*`>=0])` — нескомпрометированный пароль.

  Для проверки, был ли пароль скомпрометирован, фреймворк обращается к службе haveibeenpwned.com (**https://haveibeenpwned.com/**).

  По умолчанию пароль считается скомпрометированным, если он был зарегистрирован хотя бы в одной утечке данных. Однако можно указать другое допустимое *количество утечек*.

Эти методы можно вызывать непосредственно у класса `Password` как статические:

```
use Illuminate\Validation\Rules\Password;
. . .
'password' => ['required', 'confirmed', Password::min(12)]
```

Каждый из этих методов в качестве результата возвращает текущий объект класса `Password`, что позволяет сцеплять их вызовы:

```
'password' => ['required', 'confirmed',
               Password::min(12)->letters()->numbers()->mixedCase()]
```

Можно указать для паролей набор правил валидации по умолчанию. Для этого следует в теле метода `boot()` провайдера `App\Providers\AppServiceProvider` записать вызов статического метода `defaults(<анонимная функция>)` класса `Password`. Задаваемая *анонимная функция* не должна принимать параметров и должна возвращать набор правил валидации паролей. Пример:

```
use Illuminate\Validation\Rules\Password;
use Illuminate\Support\Facades\App;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Password::defaults(function () {
            $pwd = Password::min(12);
            // Если сайт работает в эксплуатационном режиме,
            // накладываем на пароли дополнительные требования
            if (App::environment('production')
                $pwd->letters()->numbers()->mixedCase();
            return $pwd;
        });
    }
}
```

Указать для валидации какого-либо пароля ранее заданный набор правил по умолчанию можно вызовом у класса `Password` статического метода `defaults()` без параметров:

```
'password' => ['required', Password::defaults()]
```

Если же вызвать у класса `Password` какие-либо методы, описанные ранее, задаваемые ими правила валидации будут добавлены к набору по умолчанию:

```
'password' => ['required', Password::symbols()->uncompromized()]
```

10.2.3.2. Валидация массивов элементов управления

Для валидации массивов элементов управления применяется нотация с точками и звездочками. Точки разделяют имена, индексы и ключи массивов друг от друга, а звездочка обозначает любой индекс или ключ массива.

Вот пример списка с возможностью выбора нескольких пунктов:

```
Детали машины
<select name="spare_id[]" size="6" multiple>
    <option value="1">Заклепка</option>
    <option value="2">Винт</option>
    <option value="3">Гайка</option>
    . . .
</select>
```

Здесь POST-параметр `spare_id` будет хранить массив ключей деталей, конструкция `spare_id.0` обозначит первый элемент этого массива, а конструкция `spare_id.*` — любой его элемент. Тогда правила валидации для этого списка можно записать следующим образом:

```
$validationRules = [
    // Значением параметра spare_id должен быть массив
    'spare_id' => 'array',
    // А значениями отдельных элементов этого массива — целые числа
    'spare_id.*' => 'integer'
];
```

Пример ассоциативного массива элементов управления, в который заносятся сведения об отдельном объявлении:

```
Товар:      <input name="bb[title]">
Описание:   <textarea name="bb[content]"></textarea>
Адрес:      <textarea name="bb[address]"></textarea>
Цена:       <input name="bb[price]">
```

Набор правил для его валидации таков:

```
$validationRules = [
    'bb' = > 'array:title,content,address,price',
    'bb.title' => 'required|max:50',
    'bb.content' => 'required',
    'bb.address' => 'required',
    'bb.price' => 'required|numeric'
];
```

Пример более сложного массива элементов управления, каждый элемент которого представляет собой вложенный массив, содержащий поля для ввода названия и количества каждой детали:

```
Деталь 1: название   <input name="spares[0][name]">
Деталь 1: количество <input name="spares[0][count]">
Деталь 2: название   <input name="spares[1][name]">
Деталь 2: количество <input name="spares[1][count]">
. . .
```

Параметр `spares` будет хранить массив ассоциативных массивов, содержащих названия и количества отдельных деталей. Конструкция `spares.1` обозначит второй элемент этого массива — ассоциативный массив со сведениями о детали 2, конструкция `spares.1.name` — название детали 2. А конструкция `spares.*.count` обозначит количество любой детали. Правила валидации будут выглядеть так:

```
$validation_rules = [
    'spares' => 'array',
    'spares.*' => 'array:name,count',
    // Названия деталей должны быть строками
    'spares.*.name'  => 'sometimes|string',
    // Количество деталей должно быть указано, если было задано ее название
    'spares.*.count' => 'sometimes|integer|required_with:spares.*.name'
];
```

Иногда может потребоваться указать отдельный набор правил для каждого конкретного элемента массива. Сделать это позволяет метод `forEach(<анонимная функция>)`, вызываемый у фасада `Rule`. Задаваемая *анонимная функция* будет вызвана столько раз, сколько элементов присутствует в массиве, должна принимать в качестве параметров значение

и полное (с подстановкой реальных индексов и ключей элементов вместо литералов *) имя очередного элемента и возвращать набор правил валидации, который будет применен к этому элементу. Пример:

```
$validationRules = [
    'bb' = > 'array:title,content,address,price',
    'bb.*' => Rule::forEach(function ($value, $par) {
        if ($par == 'bb.price') {
            return 'required|numeric';
        } elseif ($par == 'bb.publish') {
            return 'boolean';
        } else {
            return 'required|string';
        }
    }),
];
```

10.2.4. Написание сообщений об ошибках ввода

Набор сообщений об ошибках ввода записывается в виде ассоциативного массива и указывается в вызове валидатора (см. *разд. 10.2.1*) или возвращается методом `messages()` формального запроса (см. *разд. 10.2.2*). Значения элементов этого массива, собственно, и задают текстовые сообщения об ошибках, а ключи могут быть записаны в одном из следующих двух форматов:

- *<имя GET- или POST-параметра>.<имя правила>* — сообщение будет относиться к параметру с заданным *именем* и выводиться при нарушении правила с указанным *именем*:

```
'title.required' => 'Введите название товара'

'spares.*.count.integer' => 'Количество должно быть целым числом',
'spares.*.count.required_with' =>
                'Раз уж ввели название детали, укажите ее количество'
```

- *<имя правила>* — сообщение будет относиться ко всем параметрам:

```
'required' => 'Обязательно занесите сюда значение'
```

В тексте сообщений об ошибках можно использовать литералы формата `:<имя параметра правила валидации>`. Вместо этого литерала будет подставлено значение параметра с указанным *именем*. Пример:

```
'name' => 'required|max:40'
. . .
'name.max' => 'Длина названия рубрики не должна превышать :max символов'
```

Вместо литерала `:attribute` будет подставляться имя GET- или POST-параметра (которое совпадает с наименованием соответствующего элемента управления):

```
'name.required' => 'Введите значение в поле :attribute'
```

Вместо литерала `:value` будет подставляться значение, занесенное в соответствующий элемент управления:

```
'price.numeric' => 'Значение :value не является числом'
```

10.2.4.1. Подстановки наименований

Проблема в том, что вместо литерала `:attribute` подставляется непосредственно имя GET- или POST-параметра. Так, если пользователь при создании объявления забудет ввести адрес, ему будет выведено не особенно информативное сообщение **Введите значение в поле title**.

Проблему можно решить, предписав Laravel выводить в сообщениях указанную строку вместо имени параметра. Для этого следует создать массив подстановок наименований, ключи элементов которого должны соответствовать именам GET- и POST-параметров, а значения будут выводиться вместо этих имен. Такой массив:

- указывается в вызовах методов `validate()`, `validateWithBag()` контроллеров и метода `make()` фасада `Validator` — в последнем параметре:

```
$substitutions = [
    'title' => 'Название товара',
    'content' => 'Содержимое',
    'address' => 'Адрес',
    'rubric_id' => 'Рубрика',
    'price' => 'Цена товара',
];
$validated = $request->validate($validationRules, $errorMessages,
                                                  $substitutions);
```

- возвращается из метода `attributes()` формального запроса:

```
class RubricCreateRequest extends FormRequest {
    . . .
    public function attributes() {
        return [
            'name' => 'Название рубрики',
            'parent_id' => 'Рубрика первого уровня'
        ];
    }
}
```

При работе с массивами элементов управления применяется точечная нотация и допускается использование литерала `*`:

```
'bb.title' => 'Название товара',

'spares.*.count' => 'Количество',
```

10.2.5. Извлечение ранее введенных данных

Если валидация не увенчается успехом, данные, введенные в веб-форму, будут сохранены в серверной сессии. Извлечь их при повторном выводе страницы с той же веб-формой можно, воспользовавшись функцией-хелпером `old()`:

```
old(<наименование элемента управления>[, <значение по умолчанию>=null])
```

Функция возвращает значение, ранее занесенное в элемент управления с заданным *наименованием*. Если такого значения в сессии нет, возвращается *значение по умолчанию*. Пример:

```
<input name="title" value="{{ old('title', $bb->title) }}">
```

При работе с массивами элементов управления применяется нотация с точками:

```
<input name="bb[title]" value="{{ old('bb.title', $bb->title) }}">
```

10.2.6. Извлечение сообщений об ошибках ввода

Чтобы получить объект, хранящий все сообщения об ошибках, следует вызвать у объекта валидатора метод `errors()`:

```
$validator = Validator::make($request->all(), $validationRules,
                                                $errorMessages);
if (validator->fails()) {
    $errors = $validator->errors();
    . . .
}
```

Объект с сообщениями об ошибках создается на основе класса `Illuminate\Support\MessageBag`. Он поддерживает следующие методы:

- `get(<`*наименование*`>)` — возвращает массив сообщений об ошибках, относящихся к элементу управления с заданным *наименованием*:

  ```
  foreach ($errors->get('title') as $message) {
      . . .
  }
  . . .
  foreach ($errors->get('bb.title') as $message) {
      . . .
  }
  ```

- `first(<`*наименование*`>)` — возвращает первое сообщение об ошибке, относящееся к элементу управления с заданным *наименованием*;
- `all()` — возвращает массив сообщений об ошибках, относящихся ко всем элементам управления;
- `count()` — возвращает количество имеющихся сообщений об ошибках;
- `has(<`*наименование*`>)` — возвращает `true`, если имеются сообщения об ошибках, относящиеся к элементу управления с заданным *наименованием*, и `false` — в противном случае:

  ```
  if ($errors->has('title')) {
      . . .
  }
  ```

- `hasAny()` — возвращает `true`, если имеются сообщения об ошибках, относящиеся хотя бы к одному из элементов управления с указанными *наименованиями*, и `false` — в противном случае. Поддерживает два формата вызова:

  ```
  hasAny(<наименование 1>, <наименование 2> ... <наименование n>)
  hasAny(<массив с наименованиями>)
  ```

- `isEmpty()` — возвращает `true`, если есть хотя бы одно сообщение об ошибке, и `false` — в противном случае;

- `isNotEmpty()` — возвращает `true`, если нет ни одного сообщения об ошибке, и `false` — в противном случае;
- `any()` — то же самое, что и `isNotEmpty()`.

Описанными ранее способами можно извлечь сообщения об ошибках, находящиеся в хранилище по умолчанию. Чтобы извлечь сообщения из именованного хранилища, следует обратиться к свойству объекта сообщений об ошибках, чье имя совпадает с именем нужного хранилища:

```
// Извлекаем сообщения об ошибках из хранилища bb
foreach ($errors->bb->get('title') as $message) {
    . . .
}
```

10.2.7. Создание своих правил валидации

10.2.7.1. Правила-функции

Проще всего реализовать свое правило валидации в виде функции (*правила-функции*). Такая функция помещается непосредственно в набор правил валидации. Она должна принимать три параметра: имя GET- или POST-параметра, проверяемое значение и другую анонимную функцию, генерирующую сообщение об ошибке. В теле правила-функции следует выполнить необходимые проверки и, если они не увенчались успехом, вызвать анонимную функцию, полученную третьим параметром, передав ей строку с сообщением об ошибке.

Пример правила-функции, проверяющего, начинается ли переданное ей название рубрики с прописной буквы:

```
use Illuminate\Support\Str;
. . .
$validationRules = ['name' => [
    'required',
    function ($name, $value, $fail) {
        if ($value != Str::ucfirst($value))
            $fail('Наберите название рубрики с большой буквы');
    }
]];
$validated = $request->validate($validationRules);
```

Недостатком правила-функции является то, что ее можно использовать лишь в одном месте.

10.2.7.2. Правила-расширения

Правило-расширение может быть использовано в любом действии любого контроллера.

Создание и регистрация правила-расширения выполняются в теле метода `boot()` провайдера `App\Providers\AppServiceProvider` вызовом у фасада `Validator` метода `extend()`:

```
extend(<имя правила>, <анонимная функция>[, <сообщение об ошибке>=null])
```

Имя правила валидации должно быть уникальным, не совпадающим с именами уже имеющихся правил.

Анонимная функция, реализующая правило, должна принимать четыре параметра: имя GET- или POST-параметра, проверяемое значение, массив со значениями параметров, переданных правилу, и объект валидатора. Она должна возвращать `true`, если значение корректно, и `false` — в противном случае.

Заданное в вызове метода *сообщение об ошибке* будет выводиться в том случае, если в ассоциативном массиве сообщений, передаваемом методу `validate()` (см. *разд. 10.2.1.1* и *10.2.1.2*), не было указано другое сообщение. В противном случае *сообщение* можно не задавать.

Пример правила-расширения, выполняющего ту же проверку, что и правило из *разд. 10.2.7.1*:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Support\Str;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Validator::extend('capitalized',
            function ($name, $value, $parameters, $validator) {
                return $value == Str::ucfirst($value);
            },
            'Величина :attribute должна быть набрана с большой буквы');
    }
}
```

Использовать созданное таким образом правило можно так же, как и любое правило, встроенное во фреймворк:

```
$validationRules = ['name' => 'required|capitalized'];
$validated = $request->validate($validationRules);
```

По умолчанию, если в элемент управления не занесено никакого значения и если не задано правила, требующего занесения в него значения (например, `required`), этот элемент не проходит валидацию. Чтобы создаваемое правило-расширение применялось даже к «пустому» элементу управления, следует создать его с помощью метода `extendImplicit()` фасада `Validator`, имеющего тот же формат вызова, что и метод `extend()`. Пример:

```
Validator::extendImplicit('capitalize', ... );
```

10.2.7.3. Правила-объекты

Правило-объект, в отличие от функций и расширений, впоследствии может быть использовано при программировании других сайтов.

Новое правило-объект создается командой:

```
php artisan make:rule <имя класса правила-объекта> [--implicit]
```

Командный ключ `--implicit` предписывает создать правило, которое будет применяться даже в случае, если проверяемый элемент управления «пуст».

Классы правил-объектов объявляются в пространстве имен `App\Rules` (соответствующая папка создается автоматически). Класс правила-объекта реализует интерфейс `Illuminate\Contracts\Validation\Rule` и содержит три общедоступных метода:

- конструктор — может пригодиться для получения каких-либо нужных значений посредством внедрения зависимостей (изначально «пуст»);
- `passes($attribute, $value)` — должен проверять на корректность значение, переданное в параметре `value`, и возвращать `true`, если значение корректно, и `false` — в противном случае. В параметре `attribute` передается имя проверяемого GET- или POST-параметра. Изначально «пуст»;
- `message()` — должен возвращать сообщение об ошибке (изначально возвращает англоязычную строку, сообщающую, что возникла ошибка, без конкретики).

В листинге 10.2 показан код класса правила-объекта, выполняющего ту же проверку, что и правила, рассмотренные в *разд. 10.2.7.1* и *10.2.7.2*.

Листинг 10.2. Пример класса правила-объекта

```
namespace App\Rules;
use Illuminate\Contracts\Validation\Rule;
use Illuminate\Support\Str;
class Capitalize implements Rule {
    public function __construct() { }

    public function passes($attribute, $value) {
        return $value == Str::ucfirst($value);
    }

    public function message() {
        return 'Величина :attribute должна быть набрана с большой буквы';
    }
}
```

Объект класса, реализующий такое правило, добавляется в массив с правилами валидации:

```
use App\Rules\Capitalize;
. . .
$validationRules = ['name' => ['required', new Capitalize]];
$validated = $request->validate($validationRules);
```

Если элемент управления «пуст», он не подвергается валидации. Чтобы «пустой» элемент управления подвергался валидации, следует создать класс правила-объекта, дополнительно реализующий интерфейс `Illuminate\Contracts\Validation\ImplicitRule`. Такой класс создается при указании командного ключа `--implicit`. Пример:

```
use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\ImplicitRule;
class Capitalize implements Rule, ImplicitRule { ... }
```

Если требуется в классе правила получать доступ к остальным проверяемым значениям и (или) объекту валидатора, который производит валидацию, следует сделать следующее:

- для доступа к остальным проверяемым данным — выполнить два действия:
  - реализовать дополнительно в классе правила интерфейс `Illuminate\Contracts\Validation\DataAwareRule`;
  - объявить в классе общедоступный метод `setData($data)` — который через параметр `data` получит ассоциативный массив с проверяемыми значениями. Обычно этот метод присваивает полученный массив какому-либо свойству класса;
- для доступа к текущему объекту валидатора — выполнить два действия:
  - реализовать дополнительно в классе правила интерфейс `Illuminate\Contracts\Validation\ValidatorAwareRule`;
  - объявить в классе общедоступный метод `setValidator($validator)` — который через параметр `validator` получит объект валидатора и, например, присвоит его какому-либо свойству класса.

Методы `setData()` и `setValidator()` будут вызваны фреймворком перед выполнением валидации с применением этого класса правила. Пример:

```
use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
class Capitalize implements Rule, DataAwareRule, ValidatorAwareRule {
    private $data = [];
    private $validator;

    public function setData($data) {
        $this->data = $data;
    }

    public function setValidator($validator) {
        $this->validator = $validator;
    }
    . . .
}
```

10.3. Предварительная обработка введенных данных

По умолчанию все данные, введенные пользователем и переданные в составе клиентского запроса, проходят следующую обработку:

- удаление начальных и конечных пробелов — выполняется посредником `App\Http\Middleware\TrimStrings`;
- преобразование всех «пустых» строк в значения `null` — выполняется посредником `Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull`.

Оба посредника изначально входят в состав массива из свойства `middleware` корневого модуля `App\Http\Kernel` (см. *разд. 8.1*).

Если какую-либо из этих обработок производить не требуется совсем, можно просто исключить соответствующего посредника из массива `middleware`. Однако если нужно не выполнять эту обработку в зависимости от заданных условий, можно поступить следующим образом:

- если не нужно удалять начальные и конечные пробелы:
  - у значений определенных GET- и POST-параметров — занести имена этих параметров в массив из защищенного свойства `except` посредника `TrimStrings`:

```
class TrimStrings extends Middleware {
    protected $except = [
        . . .
        'data_with_spaces'
    ];
}
```

    Изначально в этом массиве присутствуют имена параметров `current_password`, `password` и `password_confirmation` (поля ввода пароля и его подтверждения);

  - в случае, если текущий запрос удовлетворяет заданному условию, — записать необходимое условие в вызове статического метода `skipWhen()` посредника `TrimStrings`. Этот метод должен принимать в качестве параметра объект текущего запроса и возвращать значение `true`, чтобы отключить удаление начальных и конечных пробелов у всех GET- и POST-параметров, присутствующих в текущем запросе, и `false` — чтобы включить удаление пробелов. Сам вызов метода `skipWhen()` следует записать в теле метода `boot()` провайдера `AppServiceProvider`. Пример:

```
use App\Http\Middleware\TrimStrings;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        TrimStrings::skipWhen(function ($request) {
            return $request->is('/admin/*');
        });
    }
}
```

- если не нужно преобразовывать «пустые» строки в `null` в случае, когда текущий запрос удовлетворяет заданному условию, — использовать статический метод `skipWhen()` посредника `ConvertEmptyStringsToNull`, аналогичный одноименному методу посредника `TrimStrings`:

```
use Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull;
. . .
ConvertEmptyStringsToNull::skipWhen(function ($request) {
    return $request->is('/admin/*');
});
```

10.4. Вывод веб-страниц добавления, правки и удаления записей

При выводе страницы добавления записи:

- если в веб-форме не должно быть каких-либо изначальных значений — можно просто вывести «пустую» веб-форму:

```
public function create() {
    . . .
    return view('rubric-create');
}
. . .
<input name="name" value="{{ old('name') }}">
```

- если в веб-форме должны выводиться какие-либо изначальные значения — в действии контроллера, выводящем страницу, следует создать «пустую» запись и вывести ее поля в веб-форме:

```
public function create() {
    . . .
    $rubric = new Rubric;
    return view('rubric-create', ['rubric' => $rubric]);
}
. . .
<input name="name" value="{{ old('name', $rubruc->name) }}">
```

  Изначальные значения для подстановки в поля вновь созданной «пустой» записи берутся из свойства `attributes` класса модели (см. *разд. 5.3.1*). Также их можно занести в свойства модели непосредственно в коде действия. Пример:

```
public function create() {
    . . .
    $rubric = new Rubric;
    $rubric->parent_id = Rubric::whereNull('parent_id')->first();
    return view('rubric-create', ['rubric' => $rubric]);
}
```

При выводе страницы правки записи в действии контроллера следует извлечь исправляемую запись (или, как чаще всего делают, переложить эту задачу на плечи подсистемы внедрения зависимостей) и вывести поля этой записи в веб-форме:

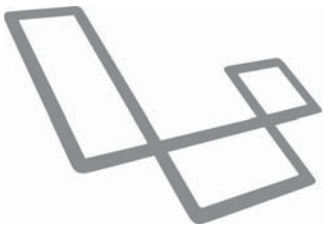
```
public function edit($id) {
    $rubric = Rubric::find($id);
    . . .
    return view('rubric-edit', ['rubric' => $rubric]);
}
. . .
<input name="name" value="{{ old('name', $rubruc->name) }}">
```

Удаление записи выполняется отправкой запроса HTTP-методом DELETE. Чтобы отправить такой запрос, на странице следует создать веб-форму с кнопкой отправки дан-

ных и вставить в эту веб-форму директиву `@method`, указав в ней метод DELETE. Эту веб-форму можно поместить на странице просмотра записи.

Однако автор рекомендует создать отдельную страницу удаления, соответствующие маршрут и действие контроллера. Пример:

```
Route::get('rubrics/{rubric}/delete', [RubricController::class, 'delete']);
. . .
public function delete($id) {
    $rubric = Rubric::find($id);
    return view('rubric-delete', ['rubric' => $rubric]);
}
. . .
<form action="{{ route('rubrics.destroy', ['rubric' => $rubric]) }}"
      method="POST">
    @csrf
    @method('DELETE')
    <input type="submit" value="Удалить">
</form>
```



ГЛАВА 11

Шаблоны: базовые инструменты

Шаблон — это образец для генерирования какой-либо веб-страницы, выдаваемой клиенту. Данные, выводящиеся на этой странице, оформляются в виде *контекста шаблона*, представляющего собой обычный ассоциативный массив PHP, на основе которого будет создан набор переменных, хранящих выводимые данные и доступных в коде шаблона. Ключи элементов этого массива зададут имена создаваемых переменных, а значения элементов будут присвоены этим переменным.

Подсистема фреймворка, выполняющая генерирование страницы на основе заданного шаблона и контекста (*рендеринг* шаблона), называется *шаблонизатором*.

Код шаблонов пишется на языке HTML с включениями особых команд — *директив* шаблонизатора. Сами шаблоны хранятся в файлах с расширением blade.php. Пути к шаблонам, записываемые в вызовах функции и метода `view()`, указываются относительно папок, в которых хранятся шаблоны (по умолчанию это папка resources\views). Для разделения имен папок и файлов в путях к шаблонам можно использовать как слеши, так и точки.

При первом рендеринге шаблона он компилируется, в результате чего присутствующие в коде шаблона директивы преобразуются в обычный PHP-код. В дальнейшем для рендеринга используются откомпилированные шаблоны — это повышает производительность. После каждого исправления шаблон компилируется повторно.

> ***ПОЛЕЗНО ЗНАТЬ***
>
> Шаблонизатор, встроенный в Laravel, носит название Blade — отсюда и расширение файлов шаблонов *blade.php*.

11.1. Настройки шаблонизатора

Настройки шаблонизатора хранятся в модуле config\view.php. Их всего две:

- `paths` — массив путей к папкам, хранящим файлы шаблонов. Изначально содержит всего один элемент `resource_path('views')`, формирующий полный путь к папке resources\views. В массив можно добавить произвольное количество путей:

```
'paths' => [
    resource_path('views'),
```

```
        base_path('special_views'),
        'c:/projects/sites/common_views'
    ],
```

- `compiled` — путь к папке, в которой будут храниться откомпилированные шаблоны.

  Значение берется из локальной настройки `VIEW_COMPILED_PATH`, изначально отсутствующей в файле .env. По умолчанию: `realpath(storage_path('framework/views'))` (формирует полный путь к папке storage\framework\views).

11.2. Директивы шаблонизатора

11.2.1. Директивы вывода данных

Для вывода данных шаблонизатор предоставляет две директивы:

- `{{ <значение> }}` — выводит заданное *значение*, которым может быть явно заданная величина, значение переменной, свойства, константы, результат выполнения функции, метода и произвольное выражение PHP:

  ```
  <h4>{{ $bb->title }}</h4>
  . . .
  <a href="{{ route('about') }}">О сайте</a>
  ```

  Перед собственно выводом все присутствующие в *значении* недопустимые символы преобразуются в соответствующие HTML-литералы:

  ```
  {{ '<h2>Дома</h2>' }}          // Результат: &lt;h2&gt;Дома&lt;/h2&gt;
  ```

  Присутствующие в *значении* HTML-литералы по умолчанию также подвергаются преобразованию:

  ```
  {{ '"&lt;br&gt;"' }}      // Результат: &quot;&amp;lt;br&amp;gt;&quot;
  ```

  Чтобы отключить преобразование HTML-литералов, достаточно в метод `boot()` провайдера `App\Providers\AppServiceProvider` поместить вызов метода `withoutDoubleEncoding()` фасада `Illuminate\Support\Facades\Blade`:

  ```
  use Illuminate\Support\Facades\Blade;
  class AppServiceProvider extends ServiceProvider {
      public function boot() {
          . . .
          Blade::withoutDoubleEncoding();
      }
  }
  . . .
  {{ '"&lt;br&gt;"' }}              // Результат: &quot;&lt;br&gt;&quot;
  ```

- `{!! <значение> !!}` — выводит заданное *значение* без преобразования недопустимых символов в литералы:

  ```
  {!! '<h2>Дома</h2>' !!}                    // Результат: <h2>Дома</h2>
  ```

11.2.1.1. Вывод данных в формате JSON

В код страниц часто помещаются веб-сценарии, обрабатывающие какие-либо данные в формате JSON. Преобразовать заданное *значение* в формат JSON и вставить его в код такого сценария позволит директива `@json(<значение>[, <настройки>[, <глубина>]])`. Для преобразования она использует функцию PHP `json_encode()`, которой передаются заданные в директиве параметры *настройки* и *глубина*. Пример:

```
<script>
    let rubric = @json($rubric);
    let bbs = @json($bbs, JSON_PRETTY_PRINT);
</script>
```

Для преобразования заданного значения в формат JSON также можно использовать статический метод `from()` фасада `Illuminate\Support\Js`, имеющий такой же формат вызова, что и директива `@json`:

```
<script>
    let rubric = Js::from($rubric);
    let bbs = Js::from($bbs, JSON_PRETTY_PRINT);
</script>
```

11.2.2. Управляющие директивы

11.2.2.1. Условные директивы и директивы выбора

- `@if ... @elseif ... @else ... @endif` — аналог условного выражения PHP:

```
@if(<условие 1>)
    <содержимое 1 — выводится, если условие 1 истинно>
[@elseif(<условие 2>)
    <содержимое 2 — выводится, если условие 2 истинно>
. . .
@elseif(<условие n>)
    <содержимое n — выводится, если условие n истинно>]
[@else
    <содержимое else — выводится, если все условия ложны>]
@endif
```

  Пример:

```
@if(count($bbs) == 0)
    <p>Объявлений нет</p>
@elseif(count($bbs) == 1)
    <p>Всего одно объявление</p>
@else
    <p>Много объявлений</p>
@endif
```

- `@unless ... @endunless` — выводит *содержимое unless*, если *условие* ложно, и *содержимое else* — в противном случае:

```
@unless(<условие>)
    <содержимое unless>
```

```
[@else
    <содержимое else>]
@endunless
```

Пример:

```
<input type="check" @unless(!$bb->publish) checked @endunless>
Публиковать объявление
```

- `@isset ... @else ... @endisset` — выводит *содержимое isset*, если все заданные *переменные* хранят значения, отличные от `null`, и *содержимое else* — в противном случае:

```
@isset(<переменная 1>, <переменная 2> ... <переменная n>)
    <содержимое isset>
[@else
    <содержимое else>]
@endisset
```

- `@empty ... @else ... @endempty` — выводит *содержимое empty*, если заданная *переменная* хранит «пустое» значение, и *содержимое else* — в противном случае. «Пустыми» считаются «пустая» строка, строка `'0'`, числа `0`, `0.0`, величины `false`, `null` и «пустой» массив. Формат записи:

```
@empty(<переменная>)
    <содержимое empty>
[@else
    <содержимое else>]
@endempty
```

Пример:

```
@empty($bbs)
    <p>Объявлений нет</p>
@else
    <p>Объявления есть</p>
@endempty
```

- `@env ... @else ... @endenv` — выводит *содержимое env*, если сайт работает в заданном *режиме* или в одном из режимов, указанных в *массиве*, и *содержимое else* — в противном случае:

```
@env(<режим работы сайта>|<массив режимов>)
    <содержимое env>
[@else
    <содержимое else>]
@endenv
```

Примеры:

```
@env('local')
    <p>Сайт работает в режиме разработки.</p>
@endenv

@env(['local', 'staging'])
    <p>Сайт работает в режиме разработки или отладки.</p>
```

```
@else
    <p>Сайт работает в режиме эксплуатации.</p>
@endenv
```

- `@production ... @else ... @endproduction` — выводит *содержимое production*, если сайт работает в эксплуатационном режиме, и *содержимое else* — в противном случае:

```
@production
    <содержимое production>
[@else
    <содержимое else>]
@endproduction
```

- `@switch ... @case ... @default ... @endswitch` — аналоги выражения выбора PHP:

```
@switch(<значение>)
    @case(<величина 1>)
        <содержимое 1 — выводится, если значение == величине 1>
        @break
    @case(<величина 2>)
        <содержимое 2 — выводится, если значение == величине 2>
        @break
    . . .
    @case(<величина n>)
        <содержимое n — выводится, если значение == величине n>
        @break
    @default
        <содержимое default — выводится, если значение != ни одной величине>
@endswitch
```

Пример:

```
@switch(Auth::user()->role)
    @case('admin')
        <p>Администратор</p>
        @break
    @case('editor')
        <p>Редактор</p>
        @break
    @case('author')
        <p>Автор объявлений</p>
        @break
    @default
        <p>Неизвестная роль</p>
@endswitch
```

11.2.2.2. Директивы циклов

- `@foreach ... @endforeach` — аналог цикла по массиву PHP:

```
@foreach(<массив> as <переменная для хранения очередного элемента>)
    <тело цикла>
@endforeach
```

Пример:

```
@foreach($rubrics as $rubric)
    <h2>{{ $rubric->name }}</h2>
@endforeach
```

- `@forelse ... @empty ... @endforelse` — аналог цикла по массиву PHP, выводящий *содержимое empty*, если *массив* «пуст»:

```
@forelse(<массив> as <переменная для хранения очередного элемента>)
    <тело цикла>
@empty
    <содержимое empty>
@endforelse
```

Пример:

```
@forelse($rubrics as $rubric)
    <h2>{{ $rubric->name }}</h2>
@empty
    <p>Рубрик нет</p>
@endforelse
```

- `@for ... @endfor` — аналог цикла со счетчиком PHP:

```
@for(<предустановка>; <условие>; <приращение>)
    <тело цикла>
@endfor
```

Пример:

```
@for($i = 1; $i < 11; $i++)
    {{ $i }} 
@endfor
```

- `@while ... @endwhile` — аналог цикла с предусловием PHP:

```
@while(<условие>)
    <тело цикла>
@endwhile
```

- `@break[(<условие>)]` — аналог оператора прерывания цикла PHP `break`. Если задано *условие*, цикл прервется, только если оно истинно. Пример:

```
@foreach($rubrics as $rubric)
    @break($rubric->id > 7)
    <h2>{{ $rubric->name }}</h2>
@endforeach
```

- `@continue[(<условие>)]` — аналог оператора прерывания текущей итерации цикла PHP `continue`. Если задано *условие*, текущая итерация цикла прервется, только если оно истинно.

Циклы можно вкладывать друг в друга:

```
@foreach($rubrics as $superrubric)
    <h2>{{ $superrubric->name }}</h2>
```

```
    @foreach($superrubric->rubrics()->get() as $subrubric)
        <h3>{{ $subrubric->name }}</h3>
    @endforeach
@endforeach
```

В теле любого цикла доступна переменная `loop`, хранящая объект с полезными сведениями о текущем цикле. Вот свойства этого объекта:

- `index` — номер текущей итерации цикла, начиная с 0;
- `iteration` — то же самое, что и `index`, но начиная с 1;
- `remaining` — количество оставшихся итераций (если это не цикл с предусловием);
- `count` — количество элементов в перебираемом в цикле массиве;
- `first` — `true`, если это первая итерация цикла, `false` — в противном случае;
- `last` — `true`, если это последняя итерация цикла, `false` — в противном случае;
- `even` — `true`, если это четная итерация цикла, `false` — в противном случае;
- `odd` — `true`, если это нечетная итерация цикла, `false` — в противном случае;
- `depth` — уровень вложенности цикла (1 — для внешнего, 2 — для вложенного в него и т. д.);
- `parent` — хранит объект со сведениями о цикле предыдущего уровня вложенности.

Пример:

```
@foreach($rubrics as $superrubric)
    @if ($loop->first) <h1>Рубрики</h1> @endif
    <h2>{{ $loop->iteration }}. {{ $superrubric->name }}</h2>
    @foreach($superrubric->rubrics()->get() as $subrubric)
        <h3>{{ $loop->parent->iteration }}.{{ $loop->iteration }}.
            {{ $subrubric->name }}</h3>
    @endforeach
@endforeach
```

11.2.3. Прочие директивы

- `{{-- <текст комментария> --}}` — вставляет в шаблон комментарий с заданным *текстом*. Этот комментарий не будет присутствовать в HTML-коде страницы, сгенерированной на основе шаблона;
- `@class(<массив стилевых классов>)` — привязывает к тегу те стилевые классы, приведенные в заданном ассоциативном *массиве*, у которых выполняются заданные условия. Ключи элементов задаваемого *массива* представляют сами стилевые классы, а значения элементов — выражения, задающие необходимые условия. Директива вставляется непосредственно в нужный тег. Пример:

```
<div @class([
    {{-- Стилевой класс is-logged-in будет привязан к блоку <div>,
         если был выполнен вход --}}
    'is-logged-in' => Auth::check(),
    {{-- Стилевой класс not-logged-in будет привязан к блоку <div>,
         если вход, наоборот, не был выполнен --}}
```

```
    'not-logged-in' => !Auth::check()
])>
```

В *массиве* также можно указать обычные, индексированные элементы — они будут привязаны к тегу в любом случае:

```
<div @class([
    'is-logged-in' => Auth::check(),
    'not-logged-in' => !Auth::check(),
    {{-- Стилевой класс p-4 будет привязан к блоку <div> в любом
         случае --}}
    'p-4'
])>
```

- `@once ... @endonce` — помещает в шаблон единственную копию заданного *содержимого* (впоследствии при попытке вставить копию того же содержимого этой же директивой ничего не произойдет):

```
@once
    <содержимое, вставляемое только один раз>
@endonce
```

Часто используется в телах директив циклов (см. *разд. 11.2.2.2*). Пример:

```
@foreach($rubrics as $rubric)
    @once
        <h3>Рубрики</h3>
    @endonce
    <h2>{{ $rubric->name }}</h2>
@endforeach
```

- `@php ... @endphp` — помещает в шаблон фрагмент *PHP-кода*:

```
@php
    <PHP-код>
@endphp
```

Пример:

```
@php $rubric_count = count($rubrics); @endphp
```

11.2.4. Запрет на обработку директив

Многие JavaScript-фреймворки, как и Laravel, используют для записи своих директив двойные фигурные скобки. Чтобы указать шаблонизатору Laravel не обрабатывать такие директивы, а оставить их как есть, достаточно поставить перед ними символы `@`. Пример:

```
{{ $rubric->name }} // Обрабатывается Laravel
@{{ rubric.name }}  // Не обрабатывается Laravel
```

Чтобы Laravel не обрабатывал управляющие директивы, принадлежащие другим фреймворкам, перед ними также нужно поставить символы `@`:

```
@@if (bb.publish)
    . . .
@@endif
```

Директива `@verbatim ... @endverbatim` указывает Laravel не обрабатывать все ее *содержимое*:

```
@verbatim
    <необрабатываемое содержимое>
@endverbatim
```

Пример:

```
@verbatim
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.price }}</p>
@endverbatim
```

Еще несколько директив шаблонизатора мы изучим далее в этой главе и последующих главах.

11.3. Вывод веб-форм и элементов управления

11.3.1. Вывод веб-форм

Внутри выводимой веб-формы следует сформировать:

- электронный жетон безопасности — с помощью директивы шаблонизатора `@csrf`:

  ```
  <form ... >
      @csrf
      . . .
  </form>
  ```

  Эта директива создает скрытое поле, хранящее электронный жетон безопасности. Получив данные, введенные в веб-форму, фреймворк сравнивает жетон, извлеченный из скрытого поля, с его копией, ранее сохраненной в серверной сессии. Если обе копии жетона идентичны, данные из веб-формы считаются заслуживающими доверия и принимаются к обработке. В противном случае Laravel полагает, что имел место случай межсайтовой атаки, и выводит сообщение с кодом статуса 419 (страница устарела).

  Формирование и проверку электронного жетона безопасности осуществляет посредник `App\Http\Middleware\VerifyCsrfToken`. Изначально он входит в группу `web` и, таким образом, связывается со всеми веб-маршрутами;

- обозначение HTTP-метода, которым выполняется клиентский запрос, если этот метод отличен от GET и POST, — директивой шаблонизатора `@method(<HTTP-метод>)`. При этом в самом теге `<form>` посредством атрибута `method` следует указать метод POST. Пример:

  ```
  <form action=" ... " method="POST">
      @csrf
      @method('DELETE')
      <input type="submit" value="Удалить">
  </form>
  ```

Вместо директивы `@method` можно использовать функцию-хелпер `method_field(<HTTP-метод>)`:

```
{{ method_field('DELETE') }}
```

11.3.2. Вывод элементов управления

При выводе элемента управления часто требуется занести в него изначальное значение: либо используемое по умолчанию, либо введенное ранее и не прошедшее валидацию, либо извлеченное из исправляемой записи. Решается эта проблема по-разному, в зависимости от типа элемента управления:

- поле ввода, регулятор, скрытое поле, область редактирования — значение выводится непосредственно вызовом функции-хелпера `old()` (см. *разд. 10.2.5*):

```
<input name="title" value="{{ old('title', $bb->title) }}">
<textarea name="content">{{ old('content', $bb->content) }}</textarea>
```

- флажок — должен быть установлен, если выводимое значение равно `true`. Для обозначения флажка как установленного применяется директива шаблонизатора `@checked`:

```
@checked(<выводимое значение>)
```

  Эта директива вставляется непосредственно в тег `<input>`, создающий флажок. Помимо этого, в тот же тег необходимо вставить атрибут `value` со значением `1`, `true`, `on` или `yes`. Пример:

```
<input type="checkbox" name="publish" value="true"
       @checked(old('publish', $bb->publish))>
```

- переключатель — должен быть установлен, если выводимое значение равно величине, представляемой переключателем. Для обозначения переключателя как установленного применяется директива `@checked(<выражение>)`. Заданное *выражение* должно проверять выводимое значение на равенство значению, записанному в атрибуте `value` тега переключателя. Директива вставляется непосредственно в тег `<input>`, создающий переключатель. Пример:

```
<input type="radio" name="type" value="buy"
       @checked(old('type', $bb->type) == 'buy')>
<input type="radio" name="type" value="sell"
       @checked(old('type', $bb->type) == 'sell')>
```

- пункт списка:
  - поддерживающего выбор только одного пункта — должен быть выбран, если выводимое значение равно величине, представляемой этим пунктом. Сделать пункт списка выбранным можно с помощью директивы шаблонизатора `@selected(<выражение>)`. Заданное *выражение* должно проверять выводимое значение на равенство значению, записанному в атрибуте `value` тега пункта. Директива вставляется непосредственно в тег `<option>`, создающий пункт списка. Пример:

```
<select name="rubric_id">
    @foreach($rubrics as $rubric)
```

```
        <option value="{{ $rubric->id }}"
          @selected(old('rubric_id', $bb->rubric_id) == $rubric->id)>
            {{ $rubric->name }}
        </option>
        @endforeach
    </select>
```

- поддерживающего выбор нескольких пунктов — должен быть выбран, если в состав выводимых значений входит величина, представляемая этим пунктом:

```
<select name="spare_id[]" size="5" multiple>
    @foreach($spares as $spare)
    <option value="{{ $spare->id }}"
            @selected(old('spare_id[]',
                      $machine->spares()->pluck('id'))
                              ->contains($spare->id))>
        {{ $spare->name }}
    </option>
    @endforeach
</select>
```

  Метод `contains()`, поддерживаемый коллекциями Laravel, возвращает `true`, если в текущей коллекции содержится элемент со значением, переданным в параметре (более подробно коллекции будут рассмотрены в *главе 15*).

Обозначить элемент управления как недоступный для ввода можно с помощью директивы шаблонизатора `@disabled(`*<условие>*`)`. Если заданное *условие* равно `true`, элемент управления станет недоступным, если `false` — доступным. Директива вставляется в тег, создающий элемент управления. Пример:

```
<input type="submit" @disabled($errors->isNotEmpty())>
```

11.3.3. Вывод сообщений об ошибках ввода

В любом шаблоне присутствует переменная `errors`, хранящая список допущенных при вводе ошибок. Этот список можно извлечь и вывести на странице. Пример:

```
<input name="title" ... >
@if($errors->has('title'))
<ul>
    @foreach ($errors->get('title') as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>
@endif
```

Также можно использовать директиву шаблонизатора `@error ... @enderror`, выводящую первое сообщение об ошибке, которая относится к элементу управления с заданным *наименованием*:

```
@error(<наименование элемента управления>[, <имя хранилища ошибок>]
    <содержимое>
@enderror
```

Если *имя хранилища ошибок* не указано, ошибки будут извлекаться из хранилища ошибок по умолчанию. Внутри *содержимого* создается переменная `message`, хранящая сообщение об ошибке. Пример:

```
<input name="title" ... >
@error('title')
    <div class="invalid-feedback">{{ $message }}</div>
@enderror
```

Директива `@error` выводит указанное в ней *содержимое*, только если имеется хоть одна ошибка ввода в элемент управления с заданным *наименованием*. Это можно использовать для вывода фрагментов содержимого, которые должны выводиться только в случае наличия ошибок. Пример привязки к полю ввода стилевого класса `is-invalid` только в случае ввода некорректного значения:

```
<input ... class="form-control @error('title') is-invalid @enderror">
```

Переменную `errors` в контекст шаблона помещает посредник `Illuminate\View\Middleware\ShareErrorsFromSession`. Изначально он входит в группу `web` и, таким образом, связывается со всеми веб-маршрутами.

11.4. Наследование шаблонов

Аналогично наследованию классов в PHP Laravel предлагает механизм *наследования шаблонов*. Базовый шаблон содержит элементы, присутствующие на всех страницах сайта: шапку, поддон, главную панель навигации, элементы разметки и пр. А производный шаблон формирует содержимое, уникальное для генерируемой им страницы: список объявлений, отдельное объявление, веб-форму добавления объявления и др.

Отдельные фрагменты уникального содержимого в производном шаблоне оформляются в виде *секций*, которых может быть произвольное количество и которые должны иметь уникальные имена. В коде базового шаблона отмечаются места, в которые должно выводиться содержимое секций с заданными именами.

По принятым соглашениям базовые шаблоны должны храниться в папке layouts. Основной базовый шаблон, используемый при генерировании большинства (или всех, если сайт несложный) страниц, опять же, по соглашениям должен иметь имя app.blade.php.

Реализовать наследование шаблонов можно двумя способами. Первый способ самый простой и пригоден в большинстве случаев:

- в коде базового шаблона — помечаются места, куда должно выводиться содержимое секций из производных шаблонов. Для этого применяется директива шаблонизатора `@yield`:

  ```
  @yield(<имя секции>[, <содержимое, выводимое, если секция не была ↵
  создана в производном шаблоне>])
  ```

  Если второй параметр не указан, в случае, когда производный шаблон не содержит секции с заданным *именем*, ничего выведено не будет.

- в коде производного шаблона:
  - указывается базовый шаблон, от которого он наследует, — директивой `@extends(<имя базового шаблона>)`. Эта директива должна находиться в первой строке кода шаблона;
  - создаются нужные секции — с помощью директивы `@section`, которая поддерживает два формата записи:

```
@section(<имя создаваемой секции>, <содержимое секции>)

@section(<имя создаваемой секции>)
    <содержимое секции>
@endsection[(<имя создаваемой секции>)]
```

    Первый формат применяется, если *содержимое секции* представляет собой короткую строку, второй — если *содержимое* занимает две и более строк. У директивы `@endsection` *имя создаваемой секции* указывать необязательно — эта возможность предусмотрена лишь для удобства программиста.

Пример:

```
{{-- Базовый шаблон layouts\app.blade.php --}}
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>@yield('title') :: Объявления</title>
    </head>
    <body>
        @yield('content')
    </body>
</html>

{{-- Производный шаблон index.blade.php --}}
@extends('layouts.app')

@section('title', 'Главная')

@section('content')
    <h1>Список объявлений</h1>
    . . .
@endsection('content')
```

При генерировании страниц в вызовах функции `view()` и одноименного метода (см. *разд. 9.5.1*) указывается производный шаблон:

```
public function index() {
    return view('index');
}
```

Второй способ реализации наследования шаблона несколько сложнее, но предлагает больше возможностей:

- в коде базового шаблона — создаются секции (той же директивой `@section`), после чего сразу же выводятся на экран (директивой `@yield`):

```
<title>@section('title') Главная @endsection @yield('title') ::
       Объявления</title>
. . .
<body>
    @section('content')
        <h1>Список объявлений</h1>
    @endsection
    @yield('content')
</body>
```

  Вместо комбинации директив `@endsection` и `@yield` можно использовать директиву `@show`:

```
<title>@section('title') Главная @show :: Объявления</title>
. . .
<body>
    @section('content')
        <h1>Список объявлений</h1>
    @show
</body>
```

- в коде произвольного шаблона:
  - указывается наследуемый базовый шаблон — директивой `@extends`;
  - создаются секции — директивой `@section`. Заданное в них содержимое полностью заменит содержимое одноименных секций из базового шаблона. Если в производном шаблоне какая-либо секция не создана, будет выведено содержимое этой секции, заданное в базовом шаблоне. Пример:

```
{{-- Содержимое секции content будет взято из производного
     шаблона, а содержимое секции title — из базового, поскольку
     в произвольном шаблоне она не создана --}}
@section('content')
    <h1>Список объявлений</h1>
    <table>
        . . .
    </table>
@endsection
```

    Если требуется добавить содержимое секции, созданной в производном шаблоне, к содержимому одноименной секции из базового шаблона, следует использовать директиву `@parent`. Будучи вставленной в содержимое секции производного шаблона, она укажет место, в которое должно быть помещено содержимое одноименной секции базового шаблона. Пример:

```
{{-- Базовый шаблон --}}
@section('content')
    <h1>Список объявлений</h1>
@show
```

```
{{-- Производный шаблон --}}
@section('content')
    @parent
    <table>
        . . .
    </table>
@endsection
```

Могут оказаться полезными следующие директивы, применяемые в базовых шаблонах:

- `@hasSection ... @else ... @endif` — выводит заданное *содержимое if*, если секция с указанным *именем* не «пуста» (имеет какое-либо содержимое, заданное в производном шаблоне), и *содержимое else* — в противном случае:

```
@hasSection(<имя секции>)
    <содержимое if — выводится, если секция с заданным именем
     не "пуста">
@else
    <содержимое else — выводится, если секция с заданным именем
     "пуста">
@endif
```

  Пример:

```
@hasSection('news')
    <h4>Новости текущего дня</h4>
    @yield('news')
@else
    <p>Новостей на сегодня нет</p>
@endif
```

- `@sectionMissing ... @else ... @endif` — выводит заданное *содержимое if*, если секция с указанным *именем*, наоборот, «пуста» (не имеет никакого содержимого, заданного в производном шаблоне), и *содержимое else* — в противном случае:

```
@sectionMissing(<имя секции>)
    <содержимое if — выводится, если секция с заданным именем "пуста">
@else
    <содержимое else — выводится, если секция с заданным именем
     не "пуста">
@endif
```

11.5. Стеки

Стек аналогичен секции — за тем исключением, что содержимое, помещаемое в стек, не заменяет уже присутствующее в нем, а дополняет его. Стеки удобно применять в производных шаблонах для задания внешних таблиц стилей и веб-сценариев, привязываемых к странице.

Для указания места шаблона, куда будет помещено содержимое стека с заданным *именем*, служит директива `@stack(<имя стека>)`.

Для добавления содержимого в стек применяются следующие директивы:

- `@push ... @endpush` — добавляет *содержимое* в конец стека с заданным *именем*:

```
@push(<имя стека>)
    <добавляемое содержимое>
@endpush
```

- `@pushOnce ... @endPushOnce` — добавляет единственную копию *содержимого* в конец стека с заданным *именем* (впоследствии при попытке добавить копию того же содержимого этой же директивой ничего не произойдет):

```
@pushOnce(<имя стека>)
    <добавляемое содержимое>
@endPushOnce
```

- `@prepend ... @endprepend` — добавляет *содержимое* в начало стека с заданным *именем*:

```
@prepend(<имя стека>)
    <добавляемое содержимое>
@endprepend
```

- `@prependOnce ... @endPrependOnce` — добавляет единственную копию *содержимого* в начало стека с заданным *именем* (впоследствии при попытке добавить копию того же содержимого этой же директивой ничего не произойдет):

```
@prependOnce(<имя стека>)
    <добавляемое содержимое>
@endPrependOnce
```

Пример:

```
{{-- Базовый шаблон --}}
@push('head')
    <link href='/css/main.js' rel="stylesheet">
@endpush
<head>
    . . .
    #stack('head')
</head>
@prependOnce('head')
    <link href='/css/controls/supercontrol.css' rel="stylesheet">
@endPrependOnce

{{-- Производный шаблон --}}
@push('head')
    <script src='/js/controls/supercontrol.js'></script>
@endpush
@prependOnce('head')
    <link href='/css/controls/supercontrol.css' rel="stylesheet">
@endPrependOnce

<!-- Будет выведено:
     <link href='/css/controls/supercontrol.css' rel="stylesheet">
```

```
<link href='/css/main.js' rel="stylesheet">
<script src='/js/controls/supercontrol.js'></script> -->
```

11.6. Включаемые шаблоны

Включаемый шаблон, как следует из названия, предназначен для вставки в другие шаблоны. В виде включаемых шаблонов обычно оформляются одинаковые фрагменты содержимого, используемые в разных шаблонах.

Будучи вставленным в обычный шаблон, включаемый шаблон получает все данные, полученные включающим его шаблоном в составе контекста.

Включаемый шаблон оформляется так же, как и обычный. По соглашениям включаемые шаблоны должны храниться в папке shared или includes.

Для вставки включаемого шаблона в обычный шаблон применяются директивы:

- `@include` — вставляет включаемый шаблон с заданным *путем*:

  ```
  @include(<путь к включаемому шаблону>[,
          <ассоциативный массив с дополнительными данными>])
  ```

  Ключи элементов *ассоциативного массива* зададут имена переменных, которые будут доступны во включаемом шаблоне, а значения элементов *массива* зададут значения этих переменных. Пример:

  ```
  {{-- Включаемый шаблон shared\errors.blade.php --}}
  @error($elname)
  <div class="invalid-feedback">{{ $message }}</div>
  @enderror
  . . .
  {{-- Обычный шаблон --}}
  <input name="title" ... >
  @include('shared.errors', ['elname' => 'title'])
  ```

  Попытка вставить несуществующий шаблон приведет к ошибке;

- `@each` — перебирает заданный *массив или коллекцию*, помещает очередной элемент в переменную с указанным *именем*, вставляет включаемый шаблон с заданным *путем* и передает ему контекст, содержащий упомянутую ранее переменную:

  ```
  @each(<путь к включаемому шаблону>, <массив или коллекция>,
        <имя переменной для хранения очередного элемента>[,
        <путь к включаемому шаблону, вставляемому, если ↵
         массив (коллекция) "пуст">])
  ```

  Если последний параметр не задан, в случае «пустого» массива (коллекции) ничего не происходит. Пример:

  ```
  @each('shared.bb', $bbs, 'bb', 'shared.bbs_empty')
  ```

- `@includeWhen` — вставляет включаемый шаблон с заданным *путем*, если результат вычисления *условия* равен `true`, в противном случае ничего не делает:

  ```
  @includeWhen(<условие>, <путь к включаемому шаблону>[,
               <ассоциативный массив с дополнительными данными>])
  ```

Пример:

```
@includeWhen(count($bbs) > 0, 'shared.bbs')
```

- `@includeUnless` — аналогична `@includeWhen`, только вставляет включаемый шаблон с заданным *путем*, если результат вычисления *условия*, напротив, равен `false`;
- `@includeIf` — аналогична `@include`, только вставляет включаемый шаблон с заданным *путем*, только если тот существует;
- `@includeFirst` — вставляет первый существующий включаемый шаблон из содержащихся в массиве:

```
@includeFirst(<массив с путями к включаемым шаблонам>[,
              <ассоциативный массив с дополнительными данными>])
```

11.6.1. Псевдонимы включаемых шаблонов

Часто используемым включаемым шаблонам можно дать короткие псевдонимы. Это выполняется в теле метода `boot()` провайдера `App\Providers\AppServiceProvider` вызовом у фасада `Illuminate\Support\Facades\Blade` метода `include()`:

```
include(<путь к включаемому шаблону>[, <псевдоним>=null])
```

Если псевдоним не указан, в качестве такового будет использован последний элемент заданного *пути* (например, шаблон с путем `shared.errors` получит псевдоним `errors`).

Для вставки включаемого шаблона по заданному псевдониму применяется директива формата:

```
@<псевдоним>[(<ассоциативный массив с дополнительными данными>)]
```

Пример явного назначения псевдонима:

```
use Illuminate\Support\Facades\Blade;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        Blade::include('shared.errors', 'errors');
    }
}
```

Использование включаемого шаблона с назначенным псевдонимом:

```
@errors(['elname' => 'title'])
```

11.7. Компоненты

Компонент — это комбинация шаблона, выводящего какой-либо фрагмент страницы, и связанной с ним программной логики, формирующей контекст этого шаблона. Компонент независим от остальной страницы (в том числе других имеющихся на ней компонентов).

Создание компонента выполняется командой:

```
php artisan make:component <имя компонента> [--inline] [--view] [--force]
```

По умолчанию создается полнофункциональный компонент (подробно о них рассказано далее).

Поддерживаются следующие командные ключи:

- `--inline` — создается бесшаблонный компонент (они будут рассмотрены далее);
- `--view` — создается бесклассовый компонент (они также будут рассмотрены далее);
- `--force` — принудительное создание компонента, если модуль с таким именем уже существует.

Пример:

```
php artisan make:component Rubrics
```

Вместо *имени компонента* можно указать путь к нему, использовав для разделения сегментов пути прямые слеши:

```
php artisan make:component Alerts/WarningAlert
```

11.7.1. Полнофункциональные компоненты

Полнофункциональный компонент содержит шаблон и связанную с ним логику, реализованную в виде класса. Такой компонент создается командой `make:component`, если ключи `--inline` и `--view` не указаны.

11.7.1.1. Создание полнофункциональных компонентов

Класс компонента объявляется в пространстве имен `App\View\Components` (соответствующая папка создается автоматически) и является производным от класса `Illuminate\View\Component`.

Если при создании компонента (см. *разд. 11.7.1*) был указан путь к нему, класс созданного таким образом компонента будет объявлен в пространстве имен, вложенном в `App\View\Components`. Например, при указании пути `Alerts/WarningAlert` компонент `WarningAlert` будет объявлен в пространстве имен `App\View\Components\Alerts`.

Шаблон компонента записывается в папку resources\views\components и имеет имя, совпадающее с именем класса компонента, в котором отдельные слова набраны в нижнем регистре через дефисы (система наименования *kebab-case*). Например, если класс компонента называется `RubricList`, то его шаблон получит имя rubric-list.blade.php).

Если класс компонента объявлен во вложенном пространстве имен, шаблон компонента будет храниться во вложенной папке, чье имя совпадает с именем вложенного пространства имен. Так, шаблон компонента `Alerts\WarningAlert` будет сохранен по пути resources\views\components\alerts\warning-alert.blade.php.

В классе компонента необходимо объявить:

- общедоступные свойства — для хранения значений, выводящихся в шаблоне;
- общедоступные методы — для вызова из кода шаблона;
- конструктор — для инициализации и получения значений атрибутов компонента (о работе с ними будет рассказано далее), а также для получения каких-либо значений через подсистему внедрения зависимостей;

- метод `render()`, общедоступный, не принимающий параметров, — для рендеринга шаблона компонента, выполняемого так же, как и рендеринг шаблона страницы (см. *разд. 9.5.1*).

Изначально в классе компонента присутствуют только «пустой» конструктор и метод `render()`, производящий рендеринг шаблона, который был сгенерирован при создании компонента.

Шаблон компонента создается с применением тех же инструментов, что и шаблон страницы. В нем доступны все общедоступные свойства и методы, объявленные в классе компонента.

В листинге 11.1 приведен код класса компонента `Rubrics`, выводящего список рубрик, а в листинге 11.2 — код его шаблона.

Листинг 11.1. Класс компонента `Rubrics`

```
namespace App\View\Components;
use Illuminate\View\Component;
use App\Models\Rubric;
class Rubrics extends Component {
    public $superrubrics = null;

    public function __construct() {
        $this->superrubrics = Rubric::whereNull('parent_id')
                                    ->orderBy('name')->get();
    }

    public function render() {
        return view('components.rubrics');
    }
}
```

Листинг 11.2. Шаблон компонента components\rubrics.blade.php

```
@foreach ($superrubrics as $spr)
    <p>{{ $spr->name }}</p>
    @foreach ($spr->rubrics()->get() as $sur)
        <p class="ms-4">
            <a href="{{ route('rubric', ['rubric' => $sur->id]) }}">
                {{ $sur->name }}
            </a>
        </p>
    @endforeach
@endforeach
```

Чтобы вызвать общедоступный метод, объявленный в классе компонента, следует обратиться к переменной, чье имя совпадает с именем этого метода. Пример вызова из шаблона общедоступного метода компонента `getSubRubrics()`:

```
class Rubrics extends Component {
    . . .
    public function getSubRubrics($superrubric) {
        return $superrubric->rubrics()->orderBy('name')->get();
    }
}
. . .
@foreach ($getSubRubrics($spr) as $sur)
    . . .
@endforeach
```

Имеется возможность сделать какие-либо общедоступные свойства и методы компонента недостижимыми из кода шаблона. Для этого достаточно объявить в классе компонента защищенное свойство `except` и присвоить ему массив с именами нужных свойств и методов. Пример:

```
class Rubrics extends Component {
    public $superrubrics = null;
    public $hiddenData2;
    public $hiddenData2;
    protected $except = ['hiddenData1', 'hiddenData2'];
    . . .
}
```

В конструкторе класса компонента можно получить требуемые для работы значения, «намекнув» об этом подсистеме внедрения зависимостей фреймворка. Например, так можно получить объект текущего запроса:

```
use Illuminate\Http\Request;
class Rubrics extends Component {
    private $request;

    public function __construct(Request $request) {
        $this->request = $request;
    }
    . . .
}
```

Для вставки созданного компонента в код шаблона применяется *тег компонента* с именем формата `x-<путь к компоненту>`. *Путь к компоненту* указывается относительно пространства имен `App\View\Components`, отдельные слова в нем набираются в нижнем регистре через дефисы, а для разделения фрагментов используется точка (вместо обратного слеша).

Тег компонента является парным. Если компонент не имеет содержимого, его тег записывается в сокращенной нотации — со слешем перед закрывающей угловой скобкой.

Пример вставки в код шаблона компонента `Rubrics`:

```
<h1>Рубрики</h1>
<x-rubrics/>
```

Для вставки компонента `Button` используется тег `<x-button/>`, для вставки компонента `RubricList` — тег `<x-rubric-list/>`, а для вставки компонента `Alerts\WarningAlert` — тег `<x-alerts.warning-alert/>`.

11.7.1.2. Передача данных в компоненты. Атрибуты компонентов

Часто бывает необходимо при выводе компонента передать ему какие-либо значения. Их можно указать в атрибутах, вставленных в тег компонента (*атрибутах компонента*). Такому атрибуту можно передать:

- постоянное значение:

```
<x-button type="primary"/>
```

- значение, являющееся результатом вычисления какого-либо выражения, — в этом случае имя атрибута следует предварить символом двоеточия (:):

```
<x-button :type="$buttonType" :caption="Str::ucfirst('добавить')"/>
```

  В этом случае выражение, вычисляющее значение атрибута, не берется в двойные фигурные скобки.

  В случае использования какого-либо JavaScript-фреймворка, в котором имена атрибутов также предваряются символом двоеточия, для предотвращения конфликтов имя атрибута, формируемого Laravel, следует предварить двойным двоеточием:

```
<x-button ::type="$buttonType" ::caption="Str::ucfirst('добавить')"/>
```

Имена атрибутов, состоящие из нескольких слов, должны записываться согласно стилю kebab-case (примеры: `button-type`, `button-caption`).

> ***В КАЧЕСТВЕ ИМЕН АТРИБУТОВ КОМПОНЕНТА НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ…***
>
> …строки `data`, `render`, `resolveView`, `shouldRender`, `view`, `withAttributes` и `withName`, поскольку сущности с такими именами используются самим Laravel.

Значения, указанные у атрибутов компонента, можно получить в конструкторе класса этого компонента через параметры, чьи имена совпадают с именами атрибутов, и, например, сохранить в свойствах для позднейшего использования:

```
class Button extends Component {
    public $type;
    public $caption;

    // Поскольку у параметра type указано значение по умолчанию,
    // одноименный атрибут не обязателен для указания
    public function __construct($caption, $type = 'primary') {
        $this->caption = $caption;
        $this->type = $type;
    }
    . . .
}
```

Атрибутам, чьи имена состоят из нескольких слов, будут соответствовать параметры с именами, записанными согласно системе camelCase:

```
public function __construct($buttonCaption, $buttonType = 'primary') {
    . . .
}
```

Полученные из атрибутов значения можно, например, вывести в шаблоне, обратившись к свойствам компонента, в которых они были сохранены:

```
<input type="submit" class="btn btn-{{ $type }}" value="{{ $caption }}">
```

У тега компонента при его выводе можно указать обычные атрибуты тегов, поддерживаемые самим языком HTML: `class`, `id`, `style` и др.:

```
<x-button caption="Сохранить" class="mt-4"/>
```

HTML-атрибуты помещаются в особое хранилище. Оно содержится в общедоступном свойстве `attributes` компонента и, как и все общедоступные свойства, доступно в шаблоне компонента. Все эти атрибуты можно поместить в какой-либо тег, содержащийся в шаблоне компонента, обратившись непосредственно к свойству `attributes`, например:

```
<div {{ $attributes }}>
    . . .
</div>
```

Объект хранилища HTML-атрибутов поддерживает ряд полезных методов:

- `get()` — возвращает значение атрибута с заданным *именем*:

  `get(<имя атрибута>[, <значение по умолчанию>=null])`

  Если атрибут с указанным именем в хранилище не существует, будет возвращено *значение по умолчанию*. Пример:

  ```
  <input type="submit"
         class="btn btn-{{ $type }} {{ $attributes->get('class') }}"
         value="{{ $caption }}">
  ```

- `only()` — возвращает новый объект хранилища, содержащий значения атрибутов с указанными *именами*:

  `only(<имя атрибута>|<массив с именами атрибутов>)`

  Пример:

  ```
  <div {{ $attributes->only(['id', 'class', 'style']) }}> ... </div>
  ```

- `except()` — возвращает новый объект хранилища, содержащий значения всех атрибутов, за исключением тех, чьи *имена* были указаны. Формат вызова такой же, как и у метода `only()`;
- `first([<значение по умолчанию>=null])` — возвращает значение первого атрибута, присутствующего в хранилище, или *значение по умолчанию*, если хранилище «пусто»;
- `whereStartsWith(<префикс>|<массив префиксов>)` — возвращает новый объект хранилища, содержащий значения атрибутов, чьи имена начинаются с заданных *префиксов*:

  ```
  <div {{ $attributes->whereStartsWith('data-')->first() }}> ...</div>
  ```

- `thatStartWith()` — то же, что и `whereStartsWith()`;
- `whereDoesntStartWith()` — возвращает новый объект хранилища, содержащий значения всех атрибутов, за исключением тех, чьи имена *не* начинаются с заданных префиксов. Формат вызова такой же, как и у метода `whereStartsWith()`;

- `filter(<анонимная функция>)` — возвращает новый объект хранилища, содержащий значения атрибутов, для которых заданная *анонимная функция* вернет значение `true`. *Анонимная функция* должна принимать два параметра: значение и имя очередного атрибута. Пример:

```
<div {{ $attributes->filter(function ($value, $key) {
    return Str::startsWith($key, 'data-') && $value != '';
} )}> ... </div>
```

- `has(<имя атрибута>)` — возвращает `true`, если в текущем хранилище присутствует атрибут с заданным *именем*, и `false` — в противном случае;
- `merge()` — возвращает новый объект хранилища, содержащий атрибуты как присутствующие в текущем хранилище, так и приведенные в заданном *ассоциативном массиве*:

```
merge(<ассоциативный массив с атрибутами>)
```

В *ассоциативном массиве* ключи элементов должны совпадать с именами атрибутов, а значения зададут значения этих атрибутов. У атрибутов, уже присутствующих в текущем хранилище, значения *не* будут заменены на указанные в *массиве*. Благодаря этой особенности метод `merge()` можно использовать для указания значений каких-либо HTML-атрибутов по умолчанию. Пример:

```
<input {{ $attributes->merge([type' => 'submit']) }}>
. . .
{{-- Выведет кнопку отправки данных с типом submit --}}
<x-button value="Отправить" />
{{-- Выведет кнопку сброса веб-формы с типом reset --}}
<x-button type="reset" value="Очистить" />
```

Исключением является атрибут `class`, у которого имеющееся и взятое из *массива* значения будут объединены в одну строку. Пример:

```
<div {{ $attributes->merge(['class' => 'container']) }}> ... </div>
. . .
{{-- Этот тег выведет: <div class="container my-3">...</div> --}}
<x-div class="my-3"> ... </div>
```

Если требуется объединить в одну строку имеющееся и какое-либо значение какого-либо атрибута, отличного от `class`, в качестве значения нужного элемента *массива*, передаваемого методу `merge()`, следует указать вызов метода `prepend()` хранилища:

```
prepend(<значение, которое будет подставлено перед имеющимся ↳
значением атрибута>)
```

Пример:

```
<div {{ $attributes->merge([
    'data-controller' => $attributes->prepend('generic-controller')
]) }}> ... </div>
. . .
<x-div data-controller="bb-controller" />
{{-- Этот тег выведет:
<div data-controller="generic-controller bb-controller">...</div> --}}
```

- `class(<массив стилевых классов>)` — аналогичен директиве шаблонизатора `@class` (см. *разд. 11.2.3*), только возвращает результат в виде нового объекта хранилища:

```
<div {{ $attributes->class([
    'container',
    'invalid-feedback' => $errors->isNotEmpty()
]) }}> ... </div>
```

11.7.1.3. Передача HTML-содержимого в компоненты. Слоты

В компонент можно передать произвольный фрагмент содержимого с целью вывести его на экран. Для этого достаточно поместить HTML-код, создающий выводимое содержимое, внутрь тега компонента, который в этом случае записывается в полной нотации — с открывающим и закрывающим тегами. Пример:

```
{{-- Компонент FormControl выведет помещенный в него тег <input> --}}
<x-form-control>
    <input name="name" value="{{ old('name', $rubric->name) }}">
</x-form-control>
```

Вставленное в компонент содержимое сохраняется в особой переменной, называемой *слотом*, имеющей имя `slot` и доступной в шаблоне компонента:

```
{{-- Шаблон компонента components\form-control.blade.php --}}
<div class="form-group">
    {{ $slot }}
</div>
```

Помимо только что описанного *слота по умолчанию* можно создать произвольное количество *именованных слотов*. Такой слот создается с помощью тега формата:

```
<x-slot name="<имя создаваемого слота>">
    <содержимое слота>
</x-slot>
```

В Laravel 9 появилась поддержка сокращенного синтаксиса создания именованных слотов:

```
<x-slot:<имя создаваемого слота>>
    <содержимое слота>
</x-slot>
```

Пример:

```
<x-form-control>
    <x-slot:label><strong>Н</strong>азвание</x-slot>
    <input name="name" value="{{ old('name', $rubric->name) }}">
</x-form-control>
```

В коде шаблона извлечь содержимое именованного слота можно из переменной, чье имя совпадает с именем слота:

```
{{-- Шаблон компонента components\form-control.blade.php --}}
<div class="form-group">
    <label>{{ $label }}</label>
```

```
    {{ $slot }}
</div>
```

Эта переменная хранит особый объект, представляющий слот.

При вставке компонента в содержимом любого слота можно обращаться к общедоступным свойствам и вызывать общедоступные методы компонента, обратившись к переменной `component`, в которой хранится его объект:

```
<x-form-control>
    <x-slot name="label">
        {{ $component->getFormattedLabel('Название') }}
    </x-slot>
    <input name="name" value="{{ old('name', $rubric->name) }}">
</x-form-control>
```

У слотов можно указывать произвольные атрибуты, как и у компонентов:

```
<x-slot:label class="highlighted-label">Н</strong>азвание</x-slot>
```

Получить хранилище этих атрибутов, аналогичное описанному в *разд. 11.7.1.2*, можно из свойства `attributes` объекта слота (который, как мы помним, доступен из переменной, имя которой совпадает с именем слота):

```
{{-- Фрагмент шаблона компонента components\form-control.blade.php --}}
<label {{ $label->attributes->get('class') }}>{{ $label }}</label>
```

11.7.2. Упрощенные компоненты

11.7.2.1. Бесшаблонные компоненты

В *бесшаблонном* компоненте отсутствует шаблон, хранящийся в отдельном файле. Вместо этого шаблон формируется и возвращается в качестве результата в методе `render()` класса контроллера. Пример:

```
class Button extends Component {
    . . .
    public function render() {
        return <<<'blade'
            <input type="submit"
                {{ $attributes->merge(['class' => 'btn btn-' . $type]) }}
                value="{{ $caption }}">
        blade;
     }
}
```

В качестве бесшаблонных имеет смысл реализовывать только компоненты с очень простым интерфейсом, но достаточно сложной программной логикой (которая и записывается в классе компонента).

Бесшаблонный компонент создается командой `make:component` утилиты artisan при указании командного ключа `--inline`.

11.7.2.2. Бесклассовые компоненты

У *бесклассового*, или *анонимного*, компонента, напротив, отсутствует класс, а имеется лишь шаблон. Соответственно, в таком компоненте невозможно реализовать сложную программную логику по выборке данных.

Тем не менее в бесклассовом компоненте можно указать, какие атрибуты компонентов он должен поддерживать. Для этого применяется директива `@props`, записываемая в самом начале кода шаблона:

```
@props(<массив с именами атрибутов компонента>)
```

В *массиве* могут присутствовать элементы двух видов:

- имя атрибута — если этот атрибут обязателен для указания;
- конструкция `<имя атрибута> => <значение атрибута по умолчанию>` — если атрибут необязателен к указанию.

В листинге 11.3 показан код бесклассового компонента components\button.blade.php, выводящего кнопку отправки данных.

Листинг 11.3. Код бесклассового компонента components\button.blade.php

```
@props(['caption', 'type' => 'primary'])

<input type="submit"
       {{ $attributes->merge(['class' => 'btn btn-' . $type]) }}
       value="{{ $caption }}">
```

Часто сложные, иерархически организованные элементы страниц формируются с помощью набора бесклассовых компонентов. Например, в иерархическом меню навигации само меню формируется одним компонентом, отдельная группа пунктов такого меню — другим, а отдельный пункт меню — третьим:

```
<x-menu menu-class="my-menu" group-class="my-group" item-class="my-item">
    <x-menu.group label="Здания">
        <x-menu.item label="Дома" />
        <x-menu.item label="Дачи" />
        <x-menu.item label="Гаражи" />
    </x-menu.group>
    <x-menu.group label="Транспорт">
        <x-menu.item label="Легковой" />
        <x-menu.item label="Грузовой" />
    </x-menu.group>
    . . .
</x-menu>
```

В таких случаях компонент верхнего уровня (в нашем примере — `Menu`) сохраняется непосредственно в папке components, а вкладываемые в него компоненты (`Group` и `Item`) — во вложенной в нее папке, имеющей то же имя, что и компонент верхнего уровня (то есть menu). Так, компоненты из нашего примера будут храниться в файлах menu.blade.php, menu\group.blade.php и menu\item.blade.php.

Однако на практике удобнее держать все компоненты, формирующие подобный элемент страницы, в одной папке. Laravel идет нам навстречу и позволяет сохранить компонент верхнего уровня в той же папке, что и остальные компоненты, дав ему имя index.blade.php. Так что наши компоненты можно поместить в одну папку, дав им имена menu\index.blade.php, menu\group.blade.php и menu\item.blade.php.

В ранее приведенном примере стилевые классы для самого меню, его отдельной группы и отдельного пункта указаны непосредственно в теге компонента `Menu`. Вставить значение этого атрибута в шаблон очень просто:

```
{{-- Шаблон компонента components\menu\index.blade.php --}}
@props(['menu-class', 'group-class', 'item-class'])

<nav class="{{ $attributes->get('menu-class') }}">
    . . .
</nav>
```

Проблема возникнет, когда в компонентах группы и пункта меню понадобится вставить в код шаблона стилевые классы, предназначенные для этих элементов. Выручит директива `@aware`, формат которой схож с таковым у директивы `@props`. В задаваемом у директивы `@aware` ассоциативном массиве следует привести атрибуты, которые были указаны у родительского компонента и к которым должен иметь доступ дочерний компонент, при необходимости указав у них значения по умолчанию. Пример:

```
{{-- Шаблон компонента components\menu\group.blade.php --}}
@aware(['group-class'])

<div class="{{ $attributes->get('group-class') }}">
    . . .
</div>

{{-- Шаблон компонента components\menu\item.blade.php --}}
@aware(['item-class'])

<div class="{{ $attributes->get('item-class') }}">
    . . .
</div>
```

> ***ДИРЕКТИВА @AWARE НЕ СМОЖЕТ ПОЛУЧИТЬ ДОСТУП К АТРИБУТУ…***
>
> …из родительского компонента, значение которого не было указано явно в теге этого компонента (что может случиться, например, если разработчик, применивший компонент, решил использовать значение этого атрибута по умолчанию). Поэтому в директиве `@aware` в дочерних компонентах у таких атрибутов следует указывать те же значения по умолчанию, что и в директиве `@props` родительского компонента.

Бесклассовый компонент создается командой `make:component` утилиты artisan при указании командного ключа `--view`.

11.7.3. Динамический компонент

Может случиться так, что в каком-либо месте страницы в одних случаях нужно выводить один компонент, а в других — другой. В таких ситуациях может помочь *динами-*

ческий компонент Laravel, выводящий на страницу компонент с указанным именем. Сам динамический компонент выводится тегом `<x-dynamic-component>`, а имя выводимого компонента указывается в его атрибуте `component`. Примеры:

```
{{-- Выводим компонент Button --}}
<x-dynamic-component component="Button" caption="Добавить"/>

{{-- Выводим компонент, чье имя хранится в переменной compName --}}
<x-dynamic-component :component="$compName"/>
```

11.8. Передача данных в шаблоны: другие способы

Самый простой способ передать данные в шаблон — оформить их в виде контекста шаблона и указать во втором параметре функции или метода `view()`. К сожалению, если несколько разных шаблонов должны выводить одно и то же значение, его тогда придется помещать в контекст каждого шаблона. Поэтому в таких случаях передавать данные в шаблоны удобнее другими способами, описываемыми далее.

11.8.1. Разделяемые значения

Разделяемое значение автоматически добавляется фреймворком в контекст *каждого* шаблона, подвергаемого рендерингу.

Разделяемые значения регистрируются в методе `boot()` провайдера `App\Providers\AppServiceProvider` (или любого другого вновь созданного — о создании своих провайдеров будет рассказано в *главе 20*). Регистрация выполняется вызовом у фасада `Illuminate\Support\Facades\View` метода `share()`, поддерживающего два формата вызова:

```
share(<имя переменной контекста шаблона>, <разделяемое значение>)
share(<ассоциативный массив разделяемых значений>)
```

Во втором формате указывается *ассоциативный массив*, ключи элементов которого зададут имена переменных контекста шаблона, которые будут хранить разделяемые значения, а значения элементов зададут сами разделяемые значения. Пример:

```
use Illuminate\Support\Facades\View;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        View::share('copyright', '© команда разработчиков');
    }
}
```

Разделяемые значения выводятся на экран так же, как обычные, — обращением к хранящим их переменным:

```
<p>{{ $copyright }}</p>
```

РАЗДЕЛЯЕМЫЕ ЗНАЧЕНИЯ РЕГИСТРИРУЮТСЯ В МОМЕНТ ИНИЦИАЛИЗАЦИИ ВЕБ-САЙТА

Изменить их впоследствии невозможно. Поэтому в качестве разделяемых значений имеет смысл использовать лишь константы и величины, вычисляемые однократно и не изменяющиеся в процессе работы сайта.

Если же требуется передать в шаблоны значения, которые могут меняться в процессе работы сайта, следует воспользоваться составителями или создателями значений (описываются далее).

11.8.2. Составители значений

Составитель значений вычисляет значения, передаваемые шаблонам, непосредственно перед их рендерингом. Также он может передать эти значения не всем имеющимся шаблонам, а лишь заданным.

Составитель значений обычно оформляется в виде класса. Он может быть объявлен в любом пространстве имен (разработчики фреймворка рекомендуют объявить его в пространстве `App\View\Composers`, создав необходимые папки самостоятельно). В составителе должны быть следующие методы:

- `compose()` — в качестве единственного параметра должен принимать объект класса `Illuminate\View\View`, представляющий подвергаемый рендерингу шаблон, и заносить в его контекст нужные значения вызовами метода `with()` (см. *разд. 9.5.1.1*);
- конструктор — только если требуется получать необходимые для работы значения посредством внедрения зависимостей.

В листинге 11.4 показан код составителя значений `App\View\Composers\RubricsComposer`, добавляющего в контекст шаблонов переменную `rubrics`, которая хранит список рубрик.

Листинг 11.4. Код составителя значений `App\View\Composers\RubricsComposer`

```
namespace App\View\Composers;
use App\Models\Rubric;
class RubricsComposer {
    public function __construct() { }

    public function compose($view) {
        $view->with('rubrics', Rubric::orderBy('name')->get());
    }
}
```

Созданный составитель значений следует зарегистрировать. Регистрация выполняется в методе `boot()` провайдера `App\Providers\AppServiceProvider` (или любого другого вновь созданного) вызовом у фасада `View` метода `composer()`:

```
composer(<обозначение шаблона>, <путь к классу составителя значений>)
```

В качестве *обозначения шаблона* можно указать:

- путь к шаблону — если задаваемые составителем значения должны быть добавлены в контекст только этого шаблона;

- массив с путями к шаблонам — если задаваемые составителем значения должны добавляться в контексты нескольких шаблонов;
- строку `'*'` — если задаваемые значения должны добавляться в контексты всех шаблонов.

Пример:

```
use Illuminate\Support\Facades\View;
use App\View\Composers\RubricsComposer;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        View::composer(['index', 'rubric'], RubricsComposer::class);
    }
}
```

Составитель значений также можно реализовать в виде анонимной функции, принимающей в качестве параметра объект шаблона. Эта функция указывается вторым параметром метода `composer()` фасада `View`. Пример:

```
View::composer('*', function ($view) {
    $view->with('rubrics', Rubric::orderBy('name')->get());
});
```

Составители значений выполняются непосредственно перед рендерингом шаблона, который производится перед самой отправкой ответа клиенту. Поэтому, если в контексте шаблона уже присутствует переменная, одноименная формируемой составителем, исходное значение этой переменной будет заменено значением, предоставленным составителем. Пример:

```
return view('index')->with(['rubrics' => $bbs]);
// В переменную rubrics контекста шаблона в конечном итоге
// будет занесен список рубрик, предоставленный составителем
```

11.8.3. Создатели значений

Создатели значений полностью аналогичны составителям, только выполняются не перед рендерингом шаблона, а непосредственно при создании представляющего его объекта вызовом функции или метода `view()`. От составителей они отличаются тремя деталями:

- обычно объявляются в пространстве имен `App\View\Creators`;
- для занесения значений в контекст шаблона применяется метод `create()`, имеющий тот же формат вызова, что и `compose()`:

```
class RubricsCreator {
    . . .
    public function create($view) {
        $view->with('rubrics', Rubric::orderBy('name')->get());
    }
}
```

- регистрация создателя значений выполняется методом `creator()` фасада `View`, имеющего тот же формат вызова, что и метод `composer()`:

```
use App\View\Creators\RubricsCreator;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        View::creator(['index', 'rubric'], RubricsCreator::class);
    }
}
```

Поскольку создатель значений выполняется при создании объектов шаблонов, есть возможность заменить предоставляемое им значение каким-либо другим, вызвав у объекта шаблона метод `with()` (см. *разд. 9.5.1.1*). Пример:

```
// В переменную rubrics контекста шаблона будет занесен список
// рубрик, заданный здесь
return view('index')->with(['rubrics' => $bbs]);
```

11.9. Обработка статических файлов

Статические файлы пересылаются клиенту без какой бы то ни было обработки. К ним относятся входящие в состав сайта внешние таблицы стилей, веб-сценарии, изображения, документы, архивы и пр.

Обработка статических файлов различается в зависимости от их местоположения:

- если статические файлы предполагается хранить непосредственно в корневой папке сайта public — их можно поместить в эту папку и записать их интернет-адреса непосредственно в HTML-коде:

```
// Будет загружен статический файл public\styles\main.css
<link href="/styles/main.css" rel="stylesheet" type="text/css">
```

- если статические файлы предполагается хранить в какой-либо папке, вложенной в папку public:
  - поместить статические файлы в эту папку;
  - в локальной настройке `ASSET_URL` или рабочей настройке `asset_url` указать путь к этой папке относительно папки public обязательно с начальным слешем:

```
ASSET_URL=/assets
```

  - использовать для генерирования интернет-адресов статических файлов функцию `asset()`:

```
asset(<путь к статическому файлу>[, <HTTPS?>=null])
```

Путь к статическому файлу указывается относительно папки, заданной в настройках `ASSET_URL` или `asset_url`. Если параметру *HTTPS* дать значение `true`, будет сгенерирован интернет-адрес, использующий протокол HTTPS, а если дать значение `false` — интернет-адрес с протоколом HTTP, если `null` — с текущим протоколом. Пример:

```
// Будет загружен статический файл public\assets\styles\main.css
<link href="{{ asset('/styles/main.css') }}" rel="stylesheet"
      type="text/css">
```

Функция `secure_asset()` генерирует интернет-адрес статического файла с протоколом HTTPS:

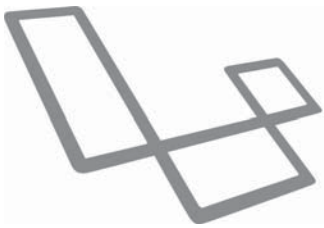
```
secure_asset(<путь к статическому файлу>)
```

- если статические файлы будут обслуживаться другим веб-сервером:
  - поместить статические файлы в корневую папку этого веб-сервера;
  - в настройках `ASSET_URL` или `asset_url` указать полный интернет-адрес этой папки:

    ```
    ASSET_URL=cdn.somesite/public
    ```

  - использовать для генерирования интернет-адресов файлов функцию `asset()`:

    ```
    // Будет загружен статический файл
    // http://cdn.somesite.ru/public/styles/main.css
    <link href="{{ asset('/styles/main.css', false) }}"
          rel="stylesheet" type="text/css">
    ```



ГЛАВА 12

Пагинация

Большие перечни каких-либо позиций (например, наборы записей, извлеченных из базы данных) при выводе практически всегда разбиваются на отдельные пронумерованные части, включающие не более определенного количества позиций, и каждая такая часть выводится на отдельной странице. Это позволяет уменьшить размер страниц и ускорить их загрузку. Для перехода на нужную часть перечня на страницах создают набор гиперссылок.

Процесс разбиения списков на части называется *пагинацией*, а занимается этим подсистема, носящая название *пагинатора*.

Laravel предоставляет три класса пагинатора: полнофункциональный, упрощенный и курсорный. Все они будут описаны далее.

12.1. Автоматическое создание пагинатора

Два метода, предоставляемых классом построителя запросов, выполняют автоматическое создание пагинатора на основе набора записей, извлеченных из базы данных. Этот пагинатор будет содержать часть с заданными порядковым *номером* и *количеством записей*:

- `paginate()` — создает полнофункциональный пагинатор.

  Полнофункциональный пагинатор позволяет переместиться не только на следующую и предыдущую, но и на произвольную часть по ее номеру. Для обозначения каждой части он использует ее порядковый номер (начиная с 1), который передается в составе интернет-адреса с определенным GET-параметром. Чтобы извлечь из базы данных все записи, входящие в состав очередной части, он отправляет СУБД SQL-команды `OFFSET` (задает порядковый номер первой извлекаемой записи) и `LIMIT` (количество извлекаемых записей).

  Формат метода:

```
paginate([<количество записей в части>=15[,
          <массив с именами извлекаемых полей>=['*'][,
          <имя GET- или POST-параметра с номером текущей ↩
           части>='page'[,
          <номер извлекаемой части>=null]]]])
```

В *массиве с именами полей* можно указать поля, значения которых следует извлечь, если же задать массив с одним элементом-звездочкой (`['*']`), будут извлечены все поля. Если *номер извлекаемой записи* не задан или равен `null`, он будет извлечен из GET- или POST-параметра с заданным *именем*. Если такого GET- или POST-параметра нет, будет извлечена первая часть.

В качестве результата метод возвращает объект класса `Illuminate\Pagination\LengthAwarePaginator`, представляющий полнофункциональный пагинатор. Примеры:

```
// Получаем пагинатор с частью, которая содержит 15 записей с полным
// набором полей. Ее номер извлекается из GET-параметра page.
$bbsPaginated = Bb::latest()->paginate();

// Получаем пагинатор с частью № 2, которая содержит 5 записей
// с полями title, content и price
$bbsPaginated = Bb::latest()->paginate(5,
                            ['title', 'content', 'price'], 'page', 2);
```

Полнофункциональный пагинатор — наиболее медленный из всех трех пагинаторов, предоставляемых Laravel, поскольку для реализации перехода на произвольную часть ему приходится вычислять общее количество позиций в перечне;

- `simplePaginate()` — создает упрощенный пагинатор.

  Упрощенный пагинатор позволяет перемещаться лишь на следующую и предыдущую части. В остальном он аналогичен полнофункциональному пагинатору.

  Формат вызова такой же, как и у метода `paginate()`. В качестве результата возвращает объект класса `Illuminate\Pagination\Paginator`.

  Упрощенный пагинатор работает несколько быстрее полнофункционального, поскольку ему не приходится вычислять общее количество позиций в перечне.

  Однако и полнофункциональный, и упрощенный пагинатор при обработке очень больших перечней существенно теряет в производительности. Это происходит потому, что у них на уровне базы данных каждая часть идентифицируется по порядковому номеру первой записи этой части, и СУБД, чтобы найти эту запись, необходимо отсчитать нужное количество записей от начала таблицы, что отнимает много времени. Например, если каждая часть содержит 20 записей и пагинатору требуется получить часть с номером 10000, которая начинается с записи номер 200000, то СУБД придется отсчитать от начала таблицы целых 199999 записей.

  Поэтому для обработки больших перечней лучше применять курсорный пагинатор, демонстрирующий исключительно высокую производительность на таблицах, удовлетворяющих предъявляемым им требованиям (будут описаны далее);

- `cursorPaginator()` — создает курсорный пагинатор.

  Курсорный пагинатор для идентификации каждой части использует значение какого-либо уникального поля первой записи этой части, обычно — ключевого поля. На основе этого значения формируется закодированная строка — *курсор*, который и передается в составе интернет-адреса через особый GET-параметр. В остальном курсорный пагинатор схож с упрощенным.

Формат метода:

```
cursorPaginate([<количество записей в части>=15[,
          <массив с именами извлекаемых полей>=['*'][,
          <имя GET- или POST-параметра с курсором текущей ↩
           части>='cursor'[,
          <курсор извлекаемой части>=null]]]])
```

Если *курсор извлекаемой записи* не задан или равен `null`, он будет извлечен из GET- или POST-параметра с заданным *именем*. Если такого GET- или POST-параметра не существует, будет выбрана первая часть.

В качестве результата метод возвращает объект класса `Illuminate\Pagination\CursorPaginator`.

Курсорному пагинатору для работы необходимо, чтобы таблица, из которой он будет извлекать записи, была упорядочена по значению какого-либо уникального поля (обычно ключевого) или по уникальной совокупности значений нескольких полей. Если таблица не удовлетворяет этим требованиям, курсорный пагинатор работать не будет.

Объект пагинатора имеет функциональность коллекции, которая содержит все записи из хранящейся в пагинаторе части. Пагинатор можно перебрать в цикле как обычную коллекцию. Пример:

```
<table>
    . . .
    @foreach ($bbsPaginated as $bb)
    <tr>
        <td>{{ $bb->title }}</td>
        <td>{{ $bb->content }}</td>
        <td>{{ $bb->price }}</td>
    </tr>
    @endforeach
</table>
```

Выводимый на странице пагинатор использует для оформления CSS-фреймворк Tailwind (**https://tailwindcss.com/**). Однако можно указать Laravel использовать для этого фреймворк Bootstrap, для чего следует открыть провайдер `App\Providers\AppServiceProvider` и добавить в тело его метода `boot()` вызов одного из следующих статических методов класса `Illuminate\Pagination\Paginator`:

- `useBootstrapFive()` — для использования Bootstrap 5;
- `useBootstrapFour()` — Bootstrap 4;
- `useBootstrap()` — то же самое, что и `useBootstrapFour()`;
- `useBootstrapThree()` — Bootstrap 3.

Пример:

```
use Illuminate\Pagination\Paginator;
. . .
class AppServiceProvider extends ServiceProvider {
    . . .
```

```
    public function boot() {
        . . .
        Paginator::useBootstrapFive();
    }
}
```

Для вывода пагинатора в виде набора гиперссылок, указывающих на отдельные части, следует вызвать метод `links()` пагинатора:

```
{{ $bbsPaginated->links() }}
```

Полнофункциональный пагинатор выведет гиперссылки (приведены в порядке слева направо), указывающие на:

- предыдущую часть;
- первую и вторую части;
- несколько частей, непосредственно предшествующих текущей. Количество этих частей задается в настройках пагинатора и изначально равно 3. Вместо остальных предшествующих частей выводится многоточие. Например, если текущей является часть номер 10, будут выведены гиперссылки на части с номерами 7–9, а вместо гиперссылок на части с номерами 3–6 отобразится многоточие;
- обозначение текущей части (не является гиперссылкой);
- несколько частей, непосредственно следующих за текущей (по аналогии с предшествующими частями);
- предпоследнюю и последнюю части;
- следующую часть.

Сами гиперссылки полнофункционального пагинатора выводятся справа, а слева отображаются порядковые номера отображаемых позиций и их общее количество.

Упрощенный и курсорный пагинаторы выведут лишь гиперссылки, указывающие на предыдущую и следующую части.

12.2. Дополнительные параметры пагинатора

У пагинатора можно указать дополнительные параметры: набор GET-параметров, которые должны присутствовать в гиперссылках, количество выводимых гиперссылок, указывающих на предшествующие или следующие части, и др. Для этого все три класса пагинатора предоставляют следующие методы:

- `appends()` — добавляет к интернет-адресам гиперссылок пагинатора заданные GET-параметры. Поддерживаются два формата вызова:

  ```
  appends(<имя GET-параметра>, <значение GET-параметра>)
  appends(<массив GET-параметров>)
  ```

  В задаваемом *массиве* ключи элементов зададут имена создаваемых GET-параметров, а значения элементов укажут их значения. Пример:

  ```
  $bbsPaginated->appends('search', 'дом')
               ->appends(['sort' => 'price', 'count' => 5]);
  ```

- `withQueryString()` — добавляет в интернет-адреса гиперссылок пагинатора все GET-параметры, присутствующие в текущем интернет-адресе;
- `fragment(<якорь>)` — добавляет в интернет-адреса гиперссылок пагинатора заданный *якорь*:

```
$bbsPaginated->fragment('bbsList');
```

- `withPath(<путь>)` — задает другой *путь*, используемый в гиперссылках пагинатора.

Метод `onEachSize(<количество частей>)` задает *количество частей* пагинатора, на которые будут выводиться гиперссылки слева и справа от текущей части. Этот метод поддерживается лишь полнофункциональным и упрощенным навигаторами (причем вызывать его у упрощенного навигатора нет никакого смысла).

12.3. Настройка отображения пагинатора

По умолчанию пагинатор выводится с применением шаблонов, являющихся частью фреймворка. Можно либо переделать эти шаблоны под свои нужды, либо на их основе создать свои шаблоны:

- чтобы переделать имеющийся шаблон — сначала следует извлечь его из состава фреймворка. Для этого достаточно набрать команду:

```
php artisan vendor:publish --tag=laravel-pagination
```

  В результате в папке resources\views будет создана папка vendor\pagination со следующими шаблонами:

  - bootstrap-5.blade.php — шаблон полнофункционального пагинатора, использующий Bootstrap 5;
  - bootstrap-4.blade.php — то же самое, только с использованием Bootstrap 4;
  - default.blade.php — то же самое, только с использованием Bootstrap 3;
  - semantic-ui.blade.php — то же самое, только с использованием CSS-фреймворка Semantic UI (**https://semantic-ui.com/**);
  - tailwind.blade.php — то же самое, только с использованием Tailwind (применяется по умолчанию при выводе полнофункционального пагинатора);
  - simple-bootstrap-5.blade.php — шаблон упрощенного пагинатора, использующий Bootstrap 5;
  - simple-bootstrap-4.blade.php — то же самое, только с использованием Bootstrap 4;
  - simple-default.blade.php — то же самое, только с использованием Bootstrap 3;
  - simple-tailwind.blade.php — то же самое, только с использованием Tailwind (применяется по умолчанию при выводе упрощенного пагинатора);

  В ДАЛЬНЕЙШЕМ LARAVEL ДЛЯ ВЫВОДА ПАГИНАТОРА БУДЕТ ИСПОЛЬЗОВАТЬ ИЗВЛЕЧЕННЫЕ ШАБЛОНЫ...

  ...хранящиеся в папке *resources\views\vendor\pagination*. Следовательно, для изменения внешнего вида пагинатора достаточно исправить какой-либо из этих шаблонов.

- чтобы указать свой шаблон для вывода конкретного пагинатора — нужно использовать расширенный формат вызова метода `links()` объекта пагинатора:

```
links(<путь к шаблону>[, <ассоциативный массив с дополнительными данными, добавляемыми в контекст шаблона>=[]])
```

Путь к шаблону указывается относительно папок, в которых хранятся шаблоны. Пример:

```
{{ @bbsPaginated->links('shared.my_paginator',
   ['active_link_color' => 'red', 'other_link_color' => 'blue']) }}
```

Также можно указать любой из стандартных шаблонов, извлеченных в папку resources\views\vendor\pagination:

```
$bbs->links('vendor.pagination.default')
```

- чтобы указать свой шаблон для всех пагинаторов, выводящихся на всех страницах сайта, — следует поместить в метод `boot()` провайдера `App\Providers\AppServiceProvider` вызов одного из следующих статических методов класса `Illuminate\Pagination\Paginator`:
  - `defaultView(<путь к шаблону>)` — задает шаблон для всех полнофункциональных пагинаторов;
  - `defaultSimpleView(<путь к шаблону>)` — задает шаблон для всех упрощенных пагинаторов.

Пример:

```
use Illuminate\Pagination\Paginator;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Paginator::defaultView('vendor.pagination.default');
        Paginator::defaultSimpleView('shared.my_simple_paginator');
    }
}
```

Чтобы указать шаблоны, использующие Bootstrap, также можно использовать статические методы, описанные в *разд. 12.1*.

В контексте шаблона пагинатора присутствуют две переменные:

- `paginator` — сам объект пагинатора (полезные методы, поддерживаемые им, мы рассмотрим позже);
- `elements` — массив с элементами, хранящими гиперссылки пагинатора. Элементы этого массива бывают двух типов:
  - строка `'...'` — обозначает пропущенные гиперссылки;
  - массив — собственно гиперссылки. Ключи элементов этого массива задают порядковые номера частей, а значения элементов — интернет-адреса соответствующих частей.

Объект любого пагинатора поддерживает следующие методы, которые пригодятся при написании шаблонов:

- `hasPages()` — возвращает `true`, если в пагинаторе присутствует более одной части, и `false` — в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить пагинатор на экран;
- `onFirstPage()` — возвращает `true`, если текущая часть является первой, и `false` — в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить гиперссылку на предыдущую часть пагинатора;
- `hasMorePages()` — возвращает `true`, если в пагинаторе присутствуют следующие части, и `false` — в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить гиперссылку на следующую часть пагинатора;
- `previousPageUrl()` — возвращает интернет-адрес предыдущей части;
- `nextPageUrl()` — возвращает интернет-адрес следующей части;
- `count()` — возвращает реальное количество позиций в текущей части (может быть меньше указанного в вызове метода, создающего пагинатор, если это последняя часть);
- `items()` — возвращает объект коллекции позиций, присутствующих в текущей части пагинатора;
- `url(<`*номер или курсор части*`>)` — возвращает интернет-адрес части с указанным *номером или курсором*;
- `perPage()` — возвращает максимальное количество позиций в части (указанное в вызове метода `paginate()` или `simplePaginate()`).

Следующие методы поддерживаются только полнофункциональным и упрощенным пагинаторами:

- `currentPage()` — возвращает номер текущей части;
- `lastPage()` — возвращает порядковый номер последней части (поддерживается только полнофункциональным пагинатором);
- `total()` — возвращает полное количество позиций во всех частях пагинатора (поддерживается только полнофункциональным пагинатором);
- `firstItem()` — возвращает порядковый номер первой позиции, присутствующей в текущей части;
- `lastItem()` — возвращает порядковый номер последней позиции, присутствующей в текущей части;
- `getUrlRange(<`*номер первой части*`>, <`*номер последней части*`>)` — формирует интернет-адреса частей пагинатора с номерами, находящимися в указанном в параметрах метода диапазоне. В качестве результата возвращается ассоциативный массив, ключи элементов которого задают порядковые номера частей, а значения элементов — интернет-адреса соответствующих частей;
- `getPageName()` — возвращает имя GET- или POST-параметра, хранящего номер текущей части;
- `setPageName(<`*имя*`>)` — задает новое *имя* для GET- или POST-параметра, хранящего номер текущей части.

Следующие методы поддерживаются только объектом курсорного пагинатора:

- `onLastPage()` — возвращает `true`, если текущая часть является последней, и `false` — в противном случае. В шаблонах используется, чтобы выяснить, следует ли выводить гиперссылку на следующую часть пагинатора;
- `getCursorName()` — возвращает имя GET- или POST-параметра, хранящего курсор текущей части;
- `setCursorName(<имя>)` — задает новое *имя* для GET- или POST-параметра, хранящего курсор текущей части.

12.4. Создание пагинатора вручную

К сожалению, штатные пагинаторы Laravel работают некорректно, если в построителе запросов была указана группировка или выборка ограниченного количества записей. В этом случае следует прибегнуть к созданию объекта пагинатора вручную.

Полнофункциональный пагинатор создается конструктором класса `Illuminate\Pagination\LengthAwarePaginator`, имеющим следующий формат вызова:

```
LengthAwarePaginator(<коллекция позиций, содержащихся в текущей части>,
                     <полное количество позиций>,
                     <количество позиций в части>[,
                     <номер текущей части>=null[,
                     <ассоциативный массив с параметрами>=[]]])
```

Первым параметром нужно передать *коллекцию* только из тех позиций, которые должны присутствовать в текущей части пагинатора. Если *номер текущей части* не задан, он будет извлечен из GET- или POST-параметра, имеющего по умолчанию имя `page`. Если такого GET- или POST-параметра нет, будет выбрана первая часть.

В *ассоциативном массиве с параметрами* ключи элементов зададут наименования параметров, а значения элементов — их значения. Поддерживаются следующие параметры:

- `pageName` — имя GET-параметра, хранящего номер текущей части пагинатора;
- `query` — ассоциативный массив с GET-параметрами, добавляемыми в интернет-адреса гиперссылок пагинатора. Ключи элементов этого массива зададут имена GET-параметров, а значения элементов — их значения;
- `fragment` — якорь, добавляемый в интернет-адреса гиперссылок пагинатора.

Пример вывода пяти самых «свежих» объявлений с пагинацией по одной записи в части:

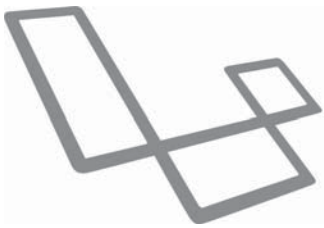
```
public function index(Request $request) {
    // Общее количество выводимых записей
    // (метод count() коллекции возвращает ее размер)
    $total = Bb::limit(5)->get()->count();
    // Номер текущей части
    $page = $request->query('page', 1);
    // Количество записей в части
    $perPage = 1;
    // Смещение от начала коллекции
    $offset = ($page - 1) * $perPage;
```

```
    // Выборка записей, которые будут присутствовать в текущей части
    $items = Bb::latest()->offset($offset)->limit($perPage)->get();
    // Создание пагинатора
    $bbs = new LengthAwarePaginator($items, $total, $perPage, $page);
    return view('index', ['bbs' => $bbs]);
}
```

Конструктор класса упрощенного пагинатора `Illuminate\Pagination\Paginator` имеет похожий формат вызова:

```
Paginator(<коллекция позиций, содержащихся в текущей части>,
          <количество позиций в части>[, <номер текущей части>=null[,
          <ассоциативный массив с параметрами>=[]]])
```

ГЛАВА 13



Разграничение доступа: базовые инструменты

Laravel предоставляет готовые инструменты для выполнения регистрации новых пользователей, входа на сайт, проверки, имеет ли текущий пользователь привилегии на выполнение какой-либо операции, выхода с сайта, а также для сброса пароля, проверки существования адреса электронной почты и подтверждения пароля.

13.1. Настройки подсистемы разграничения доступа

Настройки эти немногочисленны и хранятся в модуле config\auth.php:

- `guards` — перечень всех зарегистрированных в проекте *стражей* (модулей, обеспечивающих хранение сведений о пользователе, который выполнил вход на сайт, — *текущем пользователе*). Значением настройки является ассоциативный массив, ключи элементов которого задают имена стражей, а значениями элементов являются вложенные ассоциативные массивы с параметрами соответствующего стража. Вот эти параметры:
  - `driver` — драйвер используемого хранилища сведений о текущем пользователе. Изначально поддерживаются драйверы `session` (хранит сведения в серверной сессии) и `token` (помещает данные в электронный жетон, хранящийся на стороне клиента и пересылаемый серверу в GET-, POST-параметре или заголовке клиентского запроса);
  - `provider` — используемый *провайдер пользователей* (модуль, реализующий хранение списка зарегистрированных пользователей).

  Следующий параметр используется только драйвером `session`:

  - `remember` — время запоминания пользователя (будет описано в *разд. 13.6*) в минутах (по умолчанию: 2 628 000 минут, или ≈15 лет).

  Следующие параметры используются только драйвером `token`:

  - `input_key` — имя GET- или POST-параметра, с которым клиент будет пересылать серверу электронный жетон (по умолчанию: `api_token`);
  - `storage_key` — имя поля в таблице списка пользователей, в котором будет храниться электронный жетон (по умолчанию: `api_token`);

- `hash` — если `true`, электронный жетон перед сохранением в списке пользователей будет хешироваться с применением алгоритма SHA256, если `false` — не будет хешироваться (по умолчанию: `false`).

Изначально содержит только страж `web`, который используется при обработке веб-запросов, приходящих от веб-обозревателей, задействует драйвер `session` и провайдер пользователей `users`. Можно добавить произвольное количество своих стражей;

- `providers` — перечень зарегистрированных провайдеров пользователей. Организован так же, как и перечень стражей. Параметры провайдеров:
  - `driver` — драйвер доступа к списку зарегистрированных пользователей. Изначально поддерживаются драйверы `eloquent` (использует модель `User`) и `database` (использует построитель запросов);
  - `model` (только драйвер `eloquent`) — полный путь к классу используемой модели;
  - `table` (только драйвер `database`) — имя используемой таблицы базы данных.

  Изначально содержит провайдер пользователей `users`, использующий драйвер `eloquent` и модель `App\Models\User`. Также присутствует закомментированный код, который объявляет провайдер с тем же именем, но использующий драйвер `database` и таблицу `users`.

  Если в проекте не задействуются модели, а доступ к данным осуществляется только через построитель запросов, можно раскомментировать код, объявляющий второй провайдер `users`, не забыв закомментировать код, создающий первый провайдер `users`. Это немного повысит производительность сайта.

  Также можно добавить свои провайдеры пользователей. Это может пригодиться, если в проекте используется несколько списков зарегистрированных пользователей, хранящих пользователей, например, с разными привилегиями;

- `passwords` — перечень используемых конфигураций сброса паролей. Организован так же, как и перечни стражей и провайдеров пользователей. Параметры конфигураций:
  - `provider` — используемый провайдер пользователей;
  - `table` — имя таблицы, в которой хранятся электронные жетоны сброса паролей;
  - `expire` — промежуток времени, в течение которого сгенерированный жетон будет актуален, в минутах;
  - `throttle` — минимальный промежуток времени, в течение которого повторный доступ к странице сброса пароля будет заблокирован, в секундах.

  Изначально в перечне присутствует единственная конфигурация `users`, использующая провайдер `users`, таблицу `password_resets`, время актуальности жетона 60 минут и время блокировки повторного доступа к странице сброса 60 секунд.

  В перечень можно добавить свои конфигурации, что может пригодиться, скажем, если в проекте есть несколько провайдеров пользователей;

- `defaults` — страж и конфигурация сброса паролей по умолчанию:
  - `guard` — страж по умолчанию (изначально — `web`);
  - `passwords` — конфигурация сброса паролей по умолчанию (изначально — `users`);

- `password_timeout` — промежуток времени, в течение которого подтвержденный пароль остается актуальным, в секундах (изначально — 10 800 секунд, или три часа).

Ряд настроек указывается в контроллерах, реализующих разграничение доступа. Мы рассмотрим их позже, когда будем говорить об этих контроллерах.

13.2. Создание недостающих модулей, реализующих разграничение доступа

Для реализации разграничения доступа используется ряд программных модулей: контроллеров, провайдеров, посредников и шаблонов. Часть из них уже присутствует во вновь созданном Laravel-проекте, остальные придется создавать отдельно.

Добавить недостающие модули в проект позволит дополнительная библиотека laravel/ui. Установить ее можно командой:

```
composer require laravel/ui
```

Установив эту библиотеку, можно создать недостающие модули, набрав команду:

```
php artisan ui:auth [--views] [--force]
```

Если какой-либо модуль из создаваемых этой командой уже существует, на экране появится соответствующее предупреждение. Чтобы перезаписать этот модуль, следует ввести слово `yes`, ввод слова `no` отменяет перезапись существующего модуля.

Поддерживаются следующие командные ключи:

- `--views` — создать только шаблоны (контроллеры созданы не будут);
- `--force` — перезаписать существующие модули, не спрашивая разрешения.

Приведенная ранее команда создаст следующие модули:

- контроллеры (пути к классам указаны относительно пространства имен `App\Http\Controllers`):
  - `Auth\RegisterController` — регистрирует нового пользователя;
  - `Auth\LoginController` — выполняет вход на сайт;
  - `Auth\ForgotPasswordController` — отправляет по почтовому адресу, введенному в веб-форму, электронное письмо с гиперссылкой на страницу сброса пароля;
  - `Auth\ResetPasswordController` — выполняет сброс пароля после перехода по гиперссылке, полученной в электронном письме;
  - `Auth\VerificationController` — реализует проверку существования заданного при регистрации адреса электронной почты;
  - `Auth\ConfirmPasswordController` — реализует подтверждение пароля;
  - `HomeController` — выводит раздел пользователя;
- шаблоны (пути указаны относительно папки resources\views):
  - auth\register.blade.php — страница с веб-формой регистрации;
  - auth\login.blade.php — страница с веб-формой входа на сайт;

- auth\passwords\email.blade.php — страница с веб-формой для ввода адреса электронной почты, по которому будет отправлено письмо с гиперссылкой сброса пароля;
- auth\passwords\reset.blade.php — страница с веб-формой для собственно сброса пароля;
- auth\verify.blade.php — страница проверки существования заданного при регистрации почтового адреса;
- auth\passwords\confirm.blade.php — страница подтверждения пароля;
- home.blade.php — страница раздела пользователя;
- layouts\app.blade.php — базовый шаблон (все ранее упомянутые шаблоны являются производными от него).

В этих шаблонах присутствует секция `content`, в которой выводится основное содержимое страниц;

- database\migrations\<*отметка времени*>_create_password_resets_table.php — миграция, создает таблицу `password_resets`, хранящую электронные жетоны сброса паролей[1].

Также может пригодиться команда:

```
php artisan ui:controllers
```

Она создает только контроллеры, участвующие в реализации разграничения доступа и описанные ранее.

13.3. Маршруты, ведущие на контроллеры разграничения доступа

Помимо создания программных модулей команда `ui:auth` утилиты artisan добавляет в модуль routes\web.php, хранящий список веб-маршрутов, два выражения:

```
Auth::routes();
Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])
     ->name('home');
```

Второе выражение связывает шаблонный путь **/home** и допустимый HTTP-метод GET с действием `index()` контроллера `HomeController`, которое выводит страницу с разделом пользователя. А первое выражение содержит вызов у фасада `Illuminate\Support\Facades\Auth` метода `routes()`, который создает все остальные маршруты и имеет формат вызова:

```
routes([<ассоциативный массив с параметрами>])
```

В *ассоциативном массиве* ключи элементов должны соответствовать именам параметров, а их значения зададут значения этих параметров. Все поддерживаемые параметры имеют логический тип:

---

[1] Любопытно, что в изначально созданном «пустом» проекте уже присутствует такая же миграция. Зачем понадобилось дублировать ее в двух разных библиотеках, непонятно. Вероятно, это недосмотр разработчиков фреймворка.

- `register` — если `true`, создаются маршруты на контроллер `Auth\RegisterController` (поведение по умолчанию), если `false` — не создаются;
- `login` — если `true`, создаются маршруты на контроллер `Auth\LoginController` для выполнения входа (поведение по умолчанию), если `false` — не создаются;
- `logout` — если `true`, создается маршрут на контроллер `Auth\LoginController` для выполнения выхода (поведение по умолчанию), если `false` — не создается;
- `reset` — если `true`, создаются маршруты на контроллеры `Auth\ForgotPasswordController` и `Auth\ResetPasswordController` (поведение по умолчанию), если `false` — не создаются;
- `verify` — если `true`, создаются маршруты на контроллер `Auth\VerificationController`, если `false` — не создаются (поведение по умолчанию).

> ***Примечание***
>
> В предыдущих версиях Laravel этот параметр имел значение по умолчанию `true`.

- `confirm` — если `true`, создаются маршруты на контроллер `Auth\ConfirmPasswordController` (поведение по умолчанию), если `false` — не создаются.

Пример:

```
Auth::routes(['confirm' => false, 'verify' => true]);
```

Метод `routes()` фасада `Auth` создает группу со следующими маршрутами (записаны в формате «допустимый HTTP-метод — шаблонный путь — целевое действие контроллера-класса — имя маршрута, если указано». Под ними описано назначение действия):

Следующие два маршрута не создаются, если параметру `register` было дано значение `false`:

- GET — **/register** — `Auth\RegisterController@showRegistrationForm` — `register`.

  Вывод страницы с веб-формой регистрации нового пользователя;
- POST — **/register** — `Auth\RegisterController@register`.

  Собственно регистрация нового пользователя в списке.

Следующие два маршрута не создаются, если параметру `login` было дано значение `false`:

- GET — **/login** — `Auth\LoginController@showLoginForm` — `login`.

  Вывод страницы с веб-формой входа на сайт;
- POST — **/login** — `Auth\LoginController@login`.

  Собственно выполнение входа;

Следующий маршрут не создается, если параметру `logout` было дано значение `false`:

- POST — **/logout** — `Auth\LoginController@logout` — `logout`.

  Выполнение выхода с сайта;

Следующие четыре маршрута не создаются, если параметру `reset` было дано значение `false`:

- GET — **/password/reset** — `Auth\ForgotPasswordController@showLinkRequestForm` — `password.request`.

  Вывод страницы с веб-формой отправки электронного письма с гиперссылкой для сброса пароля;

- POST — **/password/email** — `Auth\ForgotPasswordController@sendResetLinkEmail` — `password.email`.

  Собственно отправка электронного письма с гиперссылкой для сброса пароля;

- GET — **/password/reset/{token}** — `Auth\ResetPasswordController@showResetForm` — `password.reset`.

  Вывод страницы с веб-формой сброса пароля. В URL-параметре `token` передается электронный жетон сброса пароля;

- POST — **/password/reset** — `Auth\ResetPasswordController@reset` — `password.update`.

  Собственно сброс пароля.

  Следующие три маршрута не создаются, если параметру `verify` было дано значение `false`:

- GET — **/email/verify** — `Auth\VerificationController@show` — `verification.notice`.

  Вывод страницы с веб-формой проверки существования адреса электронной почты, указанного при регистрации;

- POST — **/email/resend** — `Auth\VerificationController@resend` — `verification.resend`.

  Отправка по указанному адресу электронного письма с гиперссылкой проверки этого адреса;

- GET — **/email/verify/{id}/{hash}** — `Auth\VerificationController@verify` — `verification.verify`.

  Собственно подтверждение существования адреса электронной почты после перехода по гиперссылке, полученной в электронном письме.

  Следующие два маршрута не создаются, если параметру `confirm` было дано значение `false`:

- GET — **/password/confirm** — `Auth\ConfirmPasswordController@showConfirmForm` — `password.confirm`.

  Вывод страницы с веб-формой подтверждения пароля;

- POST — **/password/confirm** — `Auth\ConfirmPasswordController@confirm`.

  Собственно подтверждение пароля.

13.4. Служебные таблицы и модель

Во вновь созданном проекте присутствуют два модуля, имеющие отношение к хранению списка пользователей:

- <*отметка времени*>_create_users_table.php — миграция, создает таблицу `users`, хранящую список зарегистрированных пользователей. Эта таблица включает следующие поля (помимо ключевого, полей отметок создания и правки):

- `name` — строковое, длина 255 символов — регистрационное имя пользователя («логин»);
- `email` — строковое, длина 255 символов, уникальное — адрес электронной почты;
- `email_verified_at` — временна́я отметка `TIMESTAMP` — время подтверждения существования заданного адреса (необязательно для заполнения);
- `password` — строковое, длина 255 символов — хеш пароля;
- `remember_token` — электронный жетон запоминания пользователя.

При необходимости в миграцию можно добавить код, создающий дополнительные поля для хранения настоящих имени и фамилии пользователя, его роли (например: автор, редактор или администратор) и др.;

- `App\Models\User` — модель зарегистрированного пользователя. Изначально в ней поля `name`, `email` и `password` помечены как доступные для массового присваивания, а поля `password` и `remember_token` — как не подлежащие экспорту в формат JSON (подробно об этом рассказано в *главе 30*).

  Разумеется, код модели пользователя можно дополнить и исправить: объявить межтабличные связи, расширить список полей, доступных для массового присваивания, задать дополнительные свойства модели и пр.

При выполнении команды `ui:auth` утилиты artisan в проект добавляется миграция *<отметка времени>*_create_password_resets_table.php, создающая таблицу `password_resets` для записи электронных жетонов сброса паролей. В этой таблице создаются следующие поля:

- `email` — строковое, длина 255 символов, индексированное — адрес электронной почты;
- `token` — строковое, длина 255 символов — собственно электронный жетон;
- `created_at` — временна́я отметка `TIMESTAMP` — отметка создания (необязательно для заполнения).

> ***Полезно знать...***
>
> Laravel, как и, вероятно, все современные фреймворки, хранит в списке пользователей не сами пароли, а их хеши — для повышения защищенности.

13.5. Регистрация новых пользователей

Все действия по регистрации нового пользователя выполняет контроллер `Auth\RegisterController`. Часть функциональности реализована непосредственно в нем, а остальная заимствуется из трейта `Illuminate\Foundation\Auth\RegistersUsers` (хранится в модуле vendor\laravel\ui\auth-backend\RegistersUsers.php).

В этом трейте объявлены оба действия контроллера:

- `showRegistrationForm()` — выводит страницу с веб-формой регистрации, сгенерированную на основе шаблона auth\register.blade.php.

  Изначально веб-форма содержит поля ввода для занесения регистрационного имени, адреса электронной почты, пароля, его подтверждения и кнопку отправки данных;

- `register()` — регистрирует нового пользователя. Сначала проверяет корректность введенных данных, потом заносит пользователя в список, генерирует событие `Registered` (события будут описаны в *главе 22*) и выполняет вход от имени зарегистрировавшегося пользователя.

  Далее метод пытается получить серверный ответ, сообщающий об успешной регистрации (страницу с соответствующим сообщением или перенаправление на эту страницу), вызвав метод `registered()`, и в случае успеха пересылает полученный ответ клиенту. Если метод `registered()` «пуст» (это его изначальное состояние), вызывает метод `redirectPath()`, чтобы получить интернет-адрес для перенаправления. Метод `redirectPath()`, в свою очередь, обращается за адресом перенаправления к методу `redirectTo()`, а если он не объявлен (изначальное состояние) — к свойству `redirectTo` (изначально хранит путь к разделу пользователя).

В конструкторе контроллера выполняется связывание всех его действий с посредником `guest` — таким образом, зарегистрироваться на сайте сможет только гость.

Контроллер содержит следующие полезные свойства и методы, переопределив которые можно изменить его функциональность:

- `validator(array $data)` — метод, защищенный, объявлен непосредственно в контроллере. Должен с параметром `data` принимать ассоциативный массив с полученными от посетителя данными о регистрируемом пользователе и возвращать валидатор, созданный на их основе.

  Изначально создает валидатор, задающий следующие правила для элементов управления с наименованиями:

  - `name` — обязателен для указания, строка, максимальная длина — 255 символов;
  - `email` — обязателен для указания, строка, адрес электронной почты, максимальная длина — 255 символов, не должен существовать в одноименном поле таблицы `users`;
  - `password` — обязателен для указания, строка, минимальная длина — 8 символов, должен быть подтвержден в элементе управления `password_confirmation`.

  Если планируется ввод дополнительных данных о пользователе (например, настоящего имени и фамилии), следует добавить в этот валидатор соответствующие правила. Пример:

```
class RegisterController extends Controller {
    . . .
    protected function validator(array $data) {
        return Validator::make($data, [
            'name' => ['required', 'string', 'max:255'],
            . . .
            'first_name' => ['string', 'max:30'],
            'last_name' => ['string', 'max:40']
        ]);
    }
    . . .
}
```

- `create(array $data)` — метод, защищенный, объявлен непосредственно в контроллере. Должен с параметром `data` принимать сведения о новом пользователе, введенные на странице регистрации, создавать на основе этих данных нового пользователя и возвращать представляющий его объект модели.

  Изначально возвращает объект модели `User`, в котором поля `name` и `email` заполнены значениями из одноименных элементов массива `data`, а поле `password` — хешем, вычисленным на основе пароля из элемента `password`. Для вычисления хеша применяется метод `make(<хешируемое значение>)`, вызываемый у фасада `Hash`.

  Опять же, если планируется ввод дополнительных сведений о пользователе, код метода следует дополнить выражениями, заносящими значения в соответствующие поля модели. Пример:

```
class RegisterController extends Controller {
    . . .
    protected function create(array $data) {
        return User::create([
            'name' => $data['name'],
            . . .
            'first_name' => $data['first_name'],
            'name_name' => $data['last_name']
        ]);
    }
    . . .
}
```

- `redirectTo` — свойство, защищенное, объявлено в контроллере. Задает интернет-адрес для перенаправления после успешной регистрации (изначально — значение константы `HOME` провайдера `RouteServiceProvider`);
- `redirectTo()` — метод, защищенный, изначально нигде не объявлен. Должен возвращать интернет-адрес для перенаправления. Если отсутствует, перенаправление будет выполнено по интернет-адресу из свойства `redirectTo`;
- `registered(Request $request, $user)` — метод, защищенный, объявлен в трейте `RegistersUsers`. Должен возвращать серверный ответ после успешной регистрации: какую-либо страницу (например, с сообщением об успешной регистрации) или перенаправление по произвольному адресу (например, на страницу с таким сообщением). С параметром `request` принимает объект текущего клиентского запроса, с параметром `user` — объект модели, хранящий вновь зарегистрированного пользователя. Изначально «пуст»;
- `guard()` — метод, защищенный, объявлен в трейте `RegistersUsers`. Должен возвращать страж, используемый для хранения данных о зарегистрированном пользователе. Изначально возвращает страж, заданный в настройках как используемый по умолчанию.

  Для получения стража в теле этого метода следует вызвать у фасада `Illuminate\Support\Facades\Auth` метод `guard([<имя стража>])`. Если *имя стража* не указано, возвращается страж по умолчанию. Пример:

```
class RegisterController extends Controller {
    . . .
```

```
    protected function guard() {
        return Auth::guard('my_guard');
    }
}
```

ВСЕ ОПИСАННЫЕ ЗДЕСЬ И ДАЛЕЕ КОНТРОЛЛЕРЫ И ШАБЛОНЫ МОЖНО ПЕРЕДЕЛАТЬ ПОД СВОИ НУЖДЫ...

...если того потребуют обстоятельства. Методы, объявленные в трейтах, могут быть переопределены во включающих их контроллерах.

13.6. Вход на веб-сайт

На страницу входа посетитель сайта может попасть двумя способами:

- непосредственно — перейдя по пути, по которому она находится (изначально — **/login**);
- опосредованно — при попытке попасть на страницу, закрытую от гостей.

По умолчанию идентификация пользователя выполняется по заносимым им в веб-форму адресу электронной почты и паролю.

Если пользователь выполнил подряд пять безуспешных попыток войти на сайт, страница входа будет заблокирована на минуту (это поведение по умолчанию, которое можно изменить).

Laravel также поддерживает *запоминание пользователя*, активируемое, когда пользователь, выполняя вход, установит находящийся в веб-форме входа флажок **Запомнить меня**. В этом случае сайт будет «помнить», что этот пользователь вошел на сайт, неопределенно долгое время, пока сам пользователь явно не выполнит выход с сайта.

Если запоминание пользователя активировано, сайт отправит клиенту «вечный» cookie, хранящий особый электронный жетон, который, помимо этого, записывается в поле `remember_token` таблицы списка пользователей. Когда тот же пользователь вновь посетит сайт, Laravel получит этот cookie в составе первого же клиентского запроса, извлечет из него электронный жетон и сравнит с жетонами, хранящимися в списке пользователей. Если там будет обнаружено совпадение, Laravel выполнит вход на сайт от имени пользователя, имеющего этот жетон.

Если же запоминание пользователя неактивно, сама платформа PHP «выпроводит» выполнившего вход пользователя с сайта, удалив устаревшие сессии. Удаление выполняется спустя определенное время после ухода пользователя с сайта, по умолчанию: 1440 секунд, или 24 минуты (может быть изменено в настройках PHP).

Вход на сайт выполняет контроллер `Auth\LoginController`. Он использует трейты `Illuminate\Foundation\Auth\AuthenticatesUsers` (реализует бóльшую часть функциональности, хранится в модуле vendor\laravel\ui\auth-backend\AuthenticatesUsers.php) и `Illuminate\Foundation\Auth\ThrottlesLogins` (выполняет блокировку страницы входа после исчерпания разрешенных попыток).

В трейте `AuthenticatesUsers` объявлены оба действия контроллера:

- `showLoginForm()` — выводит страницу входа, сгенерированную на основе шаблона auth\login.blade.php.

Изначально веб-форма входа содержит поля ввода для занесения адреса электронной почты и пароля, флажок **Запомнить меня**, кнопку отправки данных и гиперссылку на страницу сброса пароля;

- `login()` — выполняет вход. Сначала проверяет занесенные в веб-форму данные на корректность. Далее выясняет, не превышено ли количество допустимых попыток входа, и, если это так, генерирует событие `Lockout` и выводит сообщение об ошибке 429 (слишком много запросов).

  После этого действия пытается выполнить вход. В случае успеха сбрасывает счетчик безуспешных попыток и, если посетитель попал на страницу входа:

  - непосредственно — пытается получить серверный ответ, сообщающий об успешном входе (страницу с соответствующим сообщением или перенаправление на эту страницу), вызвав метод `authenticated()`, и в случае успеха пересылает полученный ответ клиенту. Если метод `authenticated()` «пуст» (изначальное состояние), вызывает метод `redirectPath()` (был описан в *разд. 13.5*);
  - опосредованно — выполняет перенаправление на страницу, на которую пытался попасть посетитель.

  Если вход не увенчался успехом, действие инкрементирует счетчик безуспешных попыток и повторно выводит страницу входа с соответствующим сообщением.

В конструкторе контроллера `Auth\LoginController` выполняется связывание всех его действий (кроме `logout`, о котором речь пойдет позже) с посредником `guest` — таким образом, попасть на страницу входа сможет только гость.

Изменить функциональность контроллера можно, переопределив в нем следующие свойства и методы:

- `username()` — метод, общедоступный, объявлен в трейте `AuthenticatesUsers`. Должен возвращать имя поля, по значению которого будет идентифицироваться пользователь. Изначально возвращает имя поля `email`.

  Чтобы пользователь идентифицировался более традиционным образом — по регистрационному имени (логину), следует переопределить этот метод так:

```
class LoginController extends Controller {
    . . .
    public function username() {
        return 'email';
    }
}
```

- `validateLogin(Request $request)` — метод, защищенный, изначально объявлен в трейте `AuthenticatesUsers`. С параметром `request` принимает текущий клиентский запрос. Должен производить валидацию данных, введенных в веб-форму входа и полученных с запросом. Изначально требует, чтобы адрес электронной почты и пароль были обязательно указаны и представляли собой строки;
- `redirectTo` — свойство, аналогичное одноименному из контроллера `Auth\RegisterController` (см. *разд. 13.5*);
- `redirectTo()` — метод, описан в *разд. 13.5*;

- `authenticated(Request $request, $user)` — метод, защищенный, объявлен в трейте `AuthenticatesUsers`. Должен возвращать серверный ответ после успешного входа: какую-либо страницу (например, с сообщением об успешном входе) или перенаправление по произвольному адресу (например, на страницу с таким сообщением). С параметром `request` принимает объект текущего клиентского запроса, с параметром `user` — объект модели, хранящий текущего пользователя. Изначально «пуст»;
- `maxAttempts` — свойство, защищенное, изначально нигде не объявлено. Должно хранить максимальное количество безуспешных попыток войти на сайт, после которых страница входа будет заблокирована. Если не объявлено, принимается значение 5;
- `decayMinutes` — свойство, защищенное, изначально нигде не объявлено. Должно хранить промежуток времени в минутах, на который страница входа блокируется после ряда безуспешных попыток выполнить вход. Если не объявлено, принимается значение 1 минута;
- `guard()` — метод, аналогичен одноименному из трейта `RegistersUsers` (см. *разд. 13.5*).

> ***Полезно знать...***
>
> Абсолютное большинство других фреймворков для идентификации пользователя используют его регистрационное имя (логин).

13.7. Раздел пользователя

Раздел пользователя реализуется контроллером `HomeController`. Изначально он содержит всего одно действие — `index()`, выводящее страницу раздела пользователя, которая генерируется на основе шаблона home.blade.php и изначально пуста.

В конструкторе контроллера выполняется связывание всех его действий с посредником `auth` — таким образом, войти в раздел пользователя можно лишь после выполнения входа на сайт.

13.8. Собственно разграничение доступа

13.8.1. Разграничение доступа: простейшие инструменты

13.8.1.1. Разграничение доступа с помощью посредников

Проще всего ограничить доступ к определенным страницам, связав с указывающими на них маршрутами следующие посредники:

- `auth` — допускает на страницу только после выполнения входа. Если вход не был выполнен и если запрос не требует данных JSON (эта проверка выполняется вызовом метода `expectsJson()` запроса), по умолчанию выполняет перенаправление по маршруту с именем `login`. Пример:

```
Route::get('/home', [HomeController::class, 'index'])
     ->middleware('auth');
```

Обозначение этого посредника можно записать в формате `auth:<имя стража>`, если требуется указать страж, отличный от используемого по умолчанию:

```
Route::get('/home', [HomeController::class, 'index'])
     ->middleware('auth:api');
```

Под кратким обозначением `auth` «скрывается» посредник `App\Http\Middleware\Authenticate`. Переопределив объявленный в нем метод `redirectTo($request)`, можно реализовать другую логику перенаправления при попытке попасть на недоступную страницу. В параметре `request` этому методу передается объект текущего клиентского запроса;

- `guest` — наоборот, допускает на страницу только гостей, не выполнивших вход. Если вход был выполнен, по умолчанию перенаправляет на раздел пользователя. Пример:

```
use App\Http\Controllers\Auth\RegisterController;
. . .
Route::get('/register', [RegisterController::class,
                         'showRegistrationForm'])
     ->middleware('guest');
```

Обозначение `guest` имеет посредник `App\Http\Middleware\RedirectIfAuthenticated`. Он содержит метод `handle()`, исправив который можно изменить логику перенаправления в случае, если пользователь выполнил вход.

Посредника можно связать не только с маршрутом, но и с контроллером:

```
class HomeController extends Controller {
    public function __construct() {
        $this->middleware('auth');
    }
    . . .
}
```

в том числе с определенными действиями этого контроллера (подробности — в *разд. 9.1.2.4*):

```
public function __construct() {
    $this->middleware('auth')->only(['store', 'update', 'destroy']);
}
```

13.8.1.2. Разграничение доступа в шаблонах

Описанные далее директивы шаблонизатора позволяют вывести фрагмент кода в том случае, если был выполнен вход или, наоборот, если вход не был выполнен:

- `@auth ... @else ... @endauth` — выводит *содержимое if*, если был выполнен вход, и *содержимое else* — в противном случае:

```
@auth[(<имя стража>)]
    <содержимое if>
@else
    <содержимое else>
@endauth
```

Если *имя стража* не указано, используется страж по умолчанию. Пример:

```
@auth
    <form action="{{ route('logout') }}" method="POST">
        @csrf
        <input type="submit" value="Выход">
    </form>
@else
    <a href="{{ route('login') }}">Вход</a>
@endauth
```

- `@guest ... @else ... @endguest` — выводит *содержимое if*, если вход, напротив, *не* был выполнен, и *содержимое else* — в противном случае:

```
@guest[(<имя стража>)]
    <содержимое if>
@else
    <содержимое else>
@endguest
```

Если *имя стража* не указано, используется страж по умолчанию.

13.8.2. Гейты

Гейт — это функция (или метод класса), в качестве параметра принимающая объект текущего пользователя и произвольное количество других значений и определяющая, имеет ли указанный пользователь привилегии на выполнение заданной операции.

13.8.2.1. Написание гейтов

Гейты объявляются в провайдере `App\Providers\AuthServiceProvider` в методе `boot()` после вызова метода `registerPolicies()`. Объявление гейта выполняется вызовом у фасада `Illuminate\Support\Facades\Gate` метода `define()`:

```
define(<наименование выполняемой операции>, <сам гейт>)
```

Наименование выполняемой операции указывается в виде строки и должно максимально ясно описывать нужную операцию (например: `create-rubric`, `update-bb`, `delete-user`).

В качестве *гейта* можно указать одно из двух:

- анонимную функцию, принимающую произвольное количество параметров, первым из которых должен быть объект текущего пользователя. Эта функция должна возвращать `true`, если текущему пользователю разрешено выполнять запрашиваемую операцию, и `false` — в противном случае. Пример:

```
use Illuminate\Support\Facades\Gate;
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        // Этот гейт разрешает текущему пользователю править лишь те
        // объявления, автором которых он является. Вторым параметром
        // передается объект модели, хранящий исправляемое объявление.
```

```
        Gate::define('update-bb', function ($user, $bb) {
            return $user->id == $bb->user_id;
        });
    }
}
```

- массив с двумя строковыми элементами: путем к классу, в котором реализован метод-гейт, и именем этого метода:

```
namespace App\Gates;
class BbGate {
    public function deleteBb($user, $bb) {
        return $user->id == $bb->user_id;
    }
}
. . .
use Illuminate\Support\Facades\Gate;
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Gate::define('delete-bb', [App\Gates\BbGate::class,
                                   'deleteBb']);
    }
}
```

13.8.2.2. Разграничение доступа посредством гейтов

Для реализации разграничения доступа на основе гейтов фасад `Gate` предоставляет ряд методов, вызываемых в контроллерах:

- `allows()` — обращается к гейту, связанному с операцией с заданным *наименованием*, возвращает `true`, если гейт разрешил выполнение этой операции, и `false` — в противном случае:

```
allows(<наименование операции>[, <параметры, передаваемые гейту>=[]])
```

  Если гейт принимает один дополнительный параметр (без учета объекта текущего пользователя), его значение можно указать во втором параметре метода `allows()` непосредственно:

```
use Illuminate\Auth\Access\AuthorizationException;
. . .
public function updateBb(Request $request, Rubric $rubric, Bb $bb) {
    if (Gate::allows('update-bb', $bb)) {
        $bb->fill($request->all());
        . . .
    } else {
        abort(403, 'Вы не можете исправить чужое объявление');
    }
}
```

  Если же гейт принимает более двух дополнительных параметров, во втором параметре метода `allows()` нужно указать массив со значениями этих параметров:

```
public function deleteBb(Request $request, Rubric $rubric, Bb $bb) {
    if (Gate::allows('delete-bb2', [$rubric, $bb])) {
        $bb->delete();
        . . .
    } else {
        abort(403, 'Вы не можете удалить чужое объявление');
    }
}
```

Если пользователю не разрешено выполнять какую-либо операцию, следует, как показано в примерах, отправить сообщение об ошибке с кодом статуса 403 (запрещенная операция). Для этого удобно использовать функции, описанные в *разд. 9.6* (например, функцию `abort()`);

- `denies()` — обращается к гейту, связанному с операцией с заданным *наименованием*, возвращает `true`, если гейт запретил выполнение этой операции, и `false` — в противном случае. Формат вызова такой же, как и у метода `allows()`. Пример:

```
public function updateBb(Request $request, Rubric $rubric, Bb $bb) {
    if (Gate::denies('update-bb', $bb))
        abort(403, 'Вы не можете исправить чужое объявление');
    $bb->fill($request->all());
    . . .
}
```

- `check()` — обращается к гейтам, связанным с операциями с заданными *наименованиями*, возвращает `true`, если *все* гейты дали разрешение, и `false` — в противном случае:

```
check(<массив с наименованиями операций>[,
      <параметры, передаваемые гейтам>=[]])
```

Пример:

```
if (Gate::check(['update-bb', 'delete-bb'], [$rubric, $bb])) {
    // «Добро» получено
}
```

- `any()` — обращается к гейтам, связанным с операциями с заданными *наименованиями*, возвращает `true`, если *хотя бы один* гейт дал разрешение, и `false` — в противном случае. Формат вызова такой же, как и у метода `check()`;
- `none()` — обращается к гейтам, связанным с операциями с заданными *наименованиями*, возвращает `true`, если *ни один* гейт не дал разрешение, и `false` — в противном случае. Формат вызова такой же, как и у метода `check()`;
- `authorize()` — обращается к гейту, связанному с операцией с заданным *наименованием*. Если гейт разрешил выполнение операции, ничего не делает, в противном случае выдает сообщение об ошибке с кодом статуса 403. Формат вызова такой же, как и у метода `allows()`. Пример:

```
public function updateBb(Request $request, Rubric $rubric, Bb $bb) {
    Gate::authorize('update-bb', $bb);
    $bb->fill($request->all());
    . . .
}
```

Чтобы вывести сообщение об ошибке, метод возбуждает исключение `Illuminate\Auth\Access\AuthorizationException`;

- `has()` — возвращает `true`, если со *всеми* операциями, *наименования* которых были указаны, связаны какие-либо гейты, и `false` — в противном случае. Поддерживаются два формата вызова:

```
has(<наименование 1>, <наименование 2>, ... <наименование n>)
has(<массив с наименованиями операций>)
```

Пример:

```
if (Gate::has('update-bb', 'delete-bb')) {
    // С операциями update-bb и delete-bb связаны гейты
}
if (!Gate::has('create-bb')) {
    // А с операцией create-bb — нет
}
```

Все приведенные здесь методы проверяют привилегии текущего пользователя. Чтобы выполнить аналогичные проверки касательно другого пользователя, сначала следует получить объект гейта, представляющего привилегии этого пользователя, вызвав у фасада `Gate` метод `forUser(<объект пользователя>)`, после чего вызвать у полученного объекта один из описанных ранее методов. Пример:

```
$anotherUser = User::firstWhere('name', 'editor');
if (Gate::forUser($anotherUser)->allows('update-bb', $bb)) {
    . . .
}
```

13.8.2.3. Перехватчики

Перехватчики — это проверки, которые выполняются перед вызовом каждого гейта или после его выполнения. *Предварительный перехватчик* вызывается перед выполнением каждого гейта и может сам решить, имеет ли пользователь право выполнять какую-либо операцию или нет. Если предварительный перехватчик сам даст или не даст пользователю привилегию выполнять операцию, запрашиваемый гейт не вызывается (поскольку и так все уже ясно). *Завершающий перехватчик* вызывается после выполнения гейта и может изменить возвращенный им результат.

Для задания перехватчиков служат два следующих метода, вызываемых у фасада `Gate`:

- `before(<предварительный перехватчик>)` — задает *предварительный перехватчик*. Последний реализуется в виде анонимной функции, принимающей в качестве параметров объект пользователя, строку с наименованием операции и массив параметров, передаваемых вызываемому гейту. Возвращать она должна:
  - `true` — если пользователю разрешено выполнять эту операцию;
  - `false` — если пользователю запрещено выполнять ее.

    В обоих этих случаях запрашиваемый гейт не вызывается;
  - `null` — разрешено или запрещено пользователю выполнять операцию, «решает» запрашиваемый гейт.

Пример:

```
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        // Этот перехватчик разрешает пользователю admin править любые
        // объявления
        Gate::before(function ($user, $operation, $parameters) {
            if ($user->name == 'admin' && $operation == 'update-bb')
                return true;
        });
    }
}
```

- `after(<завершающий перехватчик>)` — задает *завершающий перехватчик*. Он также реализуется в виде анонимной функции, принимающей в качестве параметров объект пользователя, строку с наименованием операции, результат, возвращенный гейтом (или предварительным перехватчиком, если тот был задан), и массив с параметрами, переданными запрошенному гейту. Возвращать она должна:
  - `true` — если пользователю разрешено выполнять эту операцию;
  - `false` — если пользователю запрещено выполнять ее.

    В обоих этих случаях возвращенный результат перекроет выданный запрошенным гейтом;
  - `null` — разрешить или запретить пользователю выполнять операцию, «решит» запрошенный гейт.

  Пример:

```
// Эта проверка запрещает заблокированным пользователям добавлять
// новые объявления
Gate::after(function ($user, $operation, $result, $arguments) {
    if ($user->banned && $operation == 'create-bb')
        return false;
});
```

13.8.2.4. Гейты с развернутыми ответами

Обычный гейт в качестве «ответа» выдает обычное логическое значение — `true` или `false`. Можно создать гейт, выдающий более развернутый ответ — полноценное сообщение об ошибке.

Полноценный ответ представляется объектом класса `Illuminate\Auth\Access\Response` и создается вызовом одного из двух статических методов этого класса:

- `allow()` — возвращает ответ, сообщающий, что пользователь имеет привилегию выполнять операцию;
- `deny()` — возвращает ответ, сообщающий, что у пользователя нет привилегий на выполнение операции:

```
deny([<текст сообщения об ошибке>=null[, <код статуса>=null]])
```

Если *текст сообщения* не указан или равен `null`, в возвращаемый ответ будет помещен текст по умолчанию. Если *код статуса* не указан или равен `null`, созданный ответ получит код по умолчанию 403.

Пример:

```
use Illuminate\Auth\Access\Response;
. . .
Gate::define('update-bb', function ($user, $bb) {
    if ($user->id == $bb->user_id) {
        return Response::allow();
    } else {
        return Response::deny('Вы не можете исправить чужое объявление');
    }
});
```

Методы, описанные в *разд. 13.8.2.2*, в случае применения к гейтам с развернутыми ответами работают так же, как и в случае обычных гейтов.

Метод `inspect()`, вызываемый у фасада `Gate` и имеющий тот же формат, что и метод `allows()` (см. *разд. 13.8.2.2*), возвращает объект класса `Response`, представляющий развернутый ответ. У этого объекта можно вызвать следующие методы:

- `allowed()` — возвращает `true`, если гейт дал положительный ответ, и `false` — в противном случае;
- `denied()` — возвращает `true`, если гейт дал отрицательный ответ, и `false` — в противном случае;
- `message()` — возвращает текст сообщения об ошибке, хранящийся в текущем ответе гейта;
- `code()` — возвращает код статуса, хранящийся в текущем ответе гейта.

  Пример:

  ```
  public function update(BbRequest $request, Rubric $rubric, Bb $bb) {
      $response = Gate::inspect('update-bb', $bb);
      if ($response->allowed()) {
          $bb->fill($request->all());
          . . .
      } else {
          abort($response->code(), $response->message());
      }
  }
  ```

- `authorize()` — если ответ положительный, ничего не делает, если отрицательный — выдает сообщение об ошибке с кодом статуса 403:

  ```
  Gate::inspect('update-bb', $bb)->authorize();
  $bb->fill($request->all());
  . . .
  ```

13.8.2.5. Простые гейты

Простой гейт не связывается ни с какой операцией. Такие гейты используются, чтобы выполнить какую-либо простую проверку в одном месте кода.

Простой гейт создается вызовом одного из следующих методов фасада `Gate`:

- `allowIf()` — если указанная *анонимная функция* вернет `true`, ничего не делает, если `false` — выводит сообщение об ошибке с указанными *текстом* и *кодом статуса*. Анонимная функция должна принимать единственный параметр — объект пользователя. Формат вызова:

  ```
  allowIf(<анонимная функция>[, <текст сообщения об ошибке>=null[,
                               <код статуса>=null]])
  ```

  Если не указан *текст сообщения*, будет выведено сообщение со стандартным текстом, если не указан *код статуса*, будет использован код 403. Пример:

  ```
  public function edit(Bb $bb) {
      Gate::allowIf(function ($user) use ($bb) {
          return Auth::user()->id == $bb->user_id;
      });
      return view('bb-edit', ['bb' => $bb]);
  }
  ```

- `denyIf()` — если указанная анонимная функция вернет `false`, ничего не делает, если `true` — выводит сообщение об ошибке с указанными текстом сообщения и кодом статуса. Формат вызова такой же, как и у метода `allowIf()`.

13.8.3. Политики

Политика — это более высокоуровневая разновидность гейта. Она оформляется в виде класса, методы которого и выступают в качестве гейтов, не требует обязательного объявления в провайдере `AuthServiceProvider` и вообще проще в использовании. Каждая политика связывается с определенной моделью и управляет доступом к ней.

13.8.3.1. Создание и регистрация политик

Новый класс политики создает команда:

```
php artisan make:policy <имя класса политики> [--model=<имя модели>] ↳
[--guard=<имя стража>]
```

По умолчанию создает «пустую» политику, для получения сведений о текущем пользователе использующую текущий страж.

Поддерживаются командные ключи:

- `--model` — создает политику, работающую с указанной *моделью* (подробности будут приведены чуть позже);
- `--guard` — создает политику, использующую страж с заданным *именем*.

Класс политики объявляется в пространстве имен `App\Policies` (соответствующая папка создается автоматически) и использует трейт `Illuminate\Auth\Access\HandlesAuthorization`, в котором реализована вся необходимая логика.

В классе политики объявляются общедоступные методы-гейты. В простейшем случае они должны принимать один или два параметра: первый — объект модели `User`, пред-

ставляющий текущего пользователя, второй — объект записи, с которой он собирается работать и на работу с которой следует проверить его привилегии. Пример:

```
class BbPolicy {
    // Метод-гейт с одним параметром
    public function create(User $user) { ... }

    // Метод-гейт с двумя параметрами
    public function update(User $user, Bb $bb) { ... }
}
```

Методы-гейты могут принимать дополнительные параметры, необходимые им для работы:

```
class BbPolicy {
    . . .
    // Метод-гейт с тремя параметрами
    public function delete(User $user, Bb $bb, $rubric) { ... }
}
```

Метод политики может возвращать либо логическую величину, либо развернутый ответ в виде объекта класса `Response` (см. *разд. 13.8.2.4*). Написание простейшей политики было показано в *разд. 2.5*.

При указании у команды `make:policy` ключа `--model` генерируется политика, содержащая следующие методы-гейты:

- `viewAny(User $user)` — проверяет привилегии на просмотр списка записей;
- `view(User $user, <`*модель*`> <`*параметр*`>)` — на просмотр отдельной записи;
- `create(User $user)` — на создание записи;
- `update(User $user, <`*модель*`> <`*параметр*`>)` — на правку записи;
- `delete(User $user, <`*модель*`> <`*параметр*`>)` — на удаление записи;
- `restore(User $user, <`*модель*`> <`*параметр*`>)` — на восстановление записи, ранее подвергшейся «мягкому» удалению (см. *разд. 4.1.3.2*);
- `forceDelete(User $user, <`*модель*`> <`*параметр*`>)` — на полное удаление записи в случае, если модель реализует «мягкое» удаление.

В объявление этих методов будет подставлена *модель*, указанная в параметре `--model`, и *параметр*, чье имя будет совпадать с именем *модели*, приведенным к нижнему регистру. Пример:

```
public function update(User $user, Bb $bb) { ... }
```

В политике можно объявить метод `before()`, реализующий предварительный перехватчик (завершающий перехватчик в политике создать невозможно). Этот метод должен принимать те же параметры, что и функция, передаваемая методу `before()` фасада `Gate` (см. *разд. 13.8.2.3*). Пример:

```
class BbPolicy {
    . . .
```

```
    public function before(User $user, $operation, $parameters) {
        if ($user->name == 'admin')
            return true;
    }
}
```

Если класс политики имеет имя формата *<имя класса модели>*Policy и объявлен в пространстве имен App\Models\Policies или App\Policies, регистрировать его не нужно — Laravel найдет его самостоятельно. В противном случае следует зарегистрировать политику явно.

Регистрация политики заключается в добавлении в массив, хранящийся в защищенном свойстве policies провайдера App\Providers\AuthServiceProvider, элемента формата:

<путь к классу модели> => <путь к классу связываемой с ней политики>

Пример:

```
class AuthServiceProvider extends ServiceProvider {
    protected $policies = [
        'App\Models\Bb' => 'App\AuthPolicies\BbPolicy',
    ];
    . . .
}
```

Есть возможность задать свой механизм поиска политик на основе задаваемых классов моделей. Для этого в метод boot() провайдера AuthServiceProvider следует поместить вызов метода guessPolicyNamesUsing(*<анонимная функция>*) фасада Gate. Заданная *анонимная функция* должна принимать в параметре путь к классу модели и возвращать путь к классу соответствующей ей политики. Пример:

```
use Illuminate\Support\Facades\Gate;
use Illuminate\Support\Str;
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        $this->registerPolicies();

        // На основе полученного в параметре modelName пути к модели
        // генерирует путь к политике формата
        // App\AuthPolicies\<имя модели>Policy
        // (метод replaceFirst() будет описан в главе 14)
        Gate::guessPolicyNamesUsing(function ($modelName) {
            return Str::replaceFirst('App\Models\', 'App\AuthPolicies\',
                                     $modelName) . 'Policy';
        });
    }
}
```

При поиске политики фреймворк сначала просматривает массив из свойства policies провайдера AuthServiceProvider...

...и лишь после этого начинает искать ее самостоятельно, пользуясь алгоритмом по умолчанию или заданным в вызове метода guessPolicyNamesUsing(). Поэтому для повышения производительности имеет смысл зарегистрировать все политики явно.

13.8.3.2. Разграничение доступа посредством политик

Разграничение доступа можно выполнить:

- в маршрутах — связывая их с посредником `can`, чье обозначение указывается в формате:

  `can:<имя метода-гейта>,<проверяемая запись>`

  Метод-гейт, чье *имя* указано, должен существовать в политике, связанной с моделью, привилегии на доступ к записи которой следует проверить. В качестве *проверяемой записи* указывается имя URL-параметра, через который передается ключ записи, подлежащей обработке. Фреймворк самостоятельно найдет эту запись и передаст указанному методу-гейту во втором параметре хранящий ее объект модели (используя внедрение моделей, описанное в *разд. 8.5.2*). Пример:

  ```
  Route::patch('/home/{bb}', [BbController::class, 'update'])
      ->middleware('can:update,bb');
  ```

  Если метод-гейт принимает всего один параметр, вместо *проверяемой записи* следует передать путь к классу модели:

  ```
  Route::post('/home/{bb}', [BbController::class, 'store'])
      ->middleware('can:create,App\Models\Bb');
  ```

  К сожалению, методу-гейту, принимающему более двух параметров, передать дополнительные параметры таким способом невозможно;

- в контроллерах — вызывая у объекта текущего пользователя следующие методы:

  - `can()` — возвращает `true`, если метод-гейт с указанным *именем* разрешил выполнять операцию, и `false` — в противном случае:

    `can(<имя метода-гейта>, <значение второго параметра метода-гейта>)`

    Пример:

    ```
    public function update(Request $request,Rubric $rubric, Bb $bb) {
        if ($request->user()->can('update', $bb)) {
            $bb->fill($request->all());
            . . .
        } else {
            abort(403);
        }
    }
    ```

    Если метод-гейт принимает всего один параметр, вторым параметром в вызове метода `can()` следует указать путь к классу модели:

    ```
    public function store(Request $request, Rubric $rubric) {
        if ($request->user()->can('create', Bb::class)) {
            $bb = new Bb($request->all());
            . . .
        } else {
            abort(403);
        }
    }
    ```

Если метод-гейт принимает более двух параметров, во втором параметре методу `can()` следует передать массив со значениями второго и последующих параметров:

```
public function destroy(Request $request,Rubric $rubric,Bb $bb) {
    if ($request->user()->can('delete', [$bb, $rubric])) {
        $bb->delete();
        . . .
    } else {
        abort(403);
    }
}
```

- `cant()` — возвращает `true`, если метод-гейт с указанным именем запретил выполнять операцию, и `false` — в противном случае. Формат вызова такой же, как и у метода `can()`;
- `cannot()` — то же самое, что и `cant()`.

Также можно вызвать у самого контроллера метод `authorize()`, наследуемый всеми контроллерами-классами у трейта `Illuminate\Foundation\Auth\Access\AuthorizesRequests`. Формат его вызова такой же, как и у метода `can()`. Если политика разрешила выполнять заданную операцию, метод ничего не делает, в противном случае он выдает сообщение об ошибке с кодом статуса 403. Пример:

```
public function update(BbRequest $request, Rubric $rubric, Bb $bb) {
    $this->authorize('update', $bb);
    $bb->fill($request->all());
}
```

❒ в шаблонах — применяя следующие директивы шаблонизатора:

- `@can ... @elsecan ... @else ... @endcan` — если политика разрешила пользователю выполнять *операцию 1*, будет выведено *содержимое 1*, если разрешено выполнять *операцию 2* — будет выведено *содержимое 2*, и т. д. Если не разрешено выполнение ни одной *операции*, выводится *содержимое else*. Формат записи:

```
@can(<операция 1>, <проверяемая запись>)
    <содержимое 1>
@elsecan(<операция 2>, <проверяемая запись>)
    <содержимое 2>
. . .
@elsecan(<операция n>, <проверяемая запись>)
    <содержимое n>
[@else
    <содержимое else>]
@endcan
```

Пример:

```
@can('update', $bb)
    <a href="{{ ... }}">Исправить</a>
@else
    ...
@endcan
```

Если метод-гейт принимает всего один параметр, вторым параметром в директивы `@can` и `@elsecan` следует поместить путь к классу модели:

```
@can('create', App\Models\Bb::class)
    <a href="{{ ... }}">Добавить объявление</a>
@endcan
```

Если же метод-гейт принимает более двух параметров, вторым параметром в директивы `@can` и `@elsecan` следует поместить массив со значениями второго и последующих параметров:

```
@can('delete', [$bb, $bb->rubric])
    <a href="{{ ... }}">Удалить</a>
@endcan
```

- `@canany ... @elsecanany ... @endcanany` — если политика разрешила пользователю выполнять любую операцию из указанного *массива 1*, будет выведено *содержимое 1*, если разрешено выполнять любую операцию из *массива 2* — будет выведено *содержимое 2*, и т. д.:

```
@canany(<массив 1>, <проверяемая запись>)
    <содержимое 1>
@elsecanany(<массив 2>, <проверяемая запись>)
    <содержимое 2>
. . .
@elsecanany(<массив n>, <проверяемая запись>)
    <содержимое n>
@endcanany
```

Пример:

```
@canany(['update', 'delete'], [$bb, $bb->rubric])
    <p>Правка и удаление объявлений разрешены</p>
@endcanany
```

- `@cannot ... @elsecannot ... @else ... @endcannot` — если политика запретила пользователю выполнять *операцию 1*, будет выведено *содержимое 1*, если запретила выполнять *операцию 2* — будет выведено *содержимое 2*, и т. д. Если не запрещено выполнение ни одной *операции*, выводится *содержимое else*. Формат записи:

```
@cannot(<операция 1>, <проверяемая запись>)
    <содержимое 1>
@elsecannot(<операция 2>, <проверяемая запись>)
    <содержимое 2>
. . .
@elsecannot(<операция n>, <проверяемая запись>)
    <содержимое n>
[@else
    <содержимое else>]
@endcannot
```

13.8.3.3. Разграничение доступа в ресурсных контроллерах

Вместо того чтобы писать в каждом действии ресурсного контроллера (см. *разд. 9.1.2.1*) вызовы методов: `can()`, `cant()` и `cannot()`, можно просто вставить в код его конструктора вызов метода `authorizeResource()`, вызываемого непосредственно у контролера и наследуемого им от трейта `AuthorizesRequests`:

```
authorizeResoirce(<путь к классу модели>,
        <имя URL-параметра, через который передается ключ записи>)
```

Этот метод связывает действия, обычно присутствующие в ресурсном контроллере, с определенными методами-гейтами политики, приведенными в табл. 13.1.

Таблица 13.1. *Действия ресурсного контроллера и связываемые с ними методы-гейты политики*

Действие	Метод-гейт
`index()`	`viewAny()`
`create()`	`create()`
`edit()`	`update()`
`destroy()`	`delete()`
`show()`	`view()`
`store()`	`create()`
`update()`	`update()`

Перед выполнением какого-либо действия ресурсного контроллера фреймворк автоматически вызовет соответствующий метод-гейт и далее выполнит действие лишь в том случае, если метод-гейт разрешит его выполнение.

Пример:

```
class BbController extends Controller {
    public function __construct() {
        $this->authorizeResource(Bb::class, 'bb');
    }
    . . .
}
```

13.8.4. Разграничение доступа с помощью формальных запросов

Каждый класс формального запроса (см. *разд. 10.2.2*) содержит общедоступный, не принимающий параметров метод `authorize()`. Его можно использовать для проверки, имеет ли пользователь привилегии для выполнения какой-либо операции. Метод должен возвращать `true`, если пользователь имеет необходимые привилегии, и `false` — если не имеет.

Пример проверки, имеет ли пользователь привилегии на правку и удаление выбранного объявления:

```
class BbRequest extends FormRequest {
    . . .
    public function authorize() {
        $bbId = $this->route('bb');
        if ($bbId) {
            $bb = Bb::find($bbId);
            return $request->user()->id == $bb->id;
        } else {
            return false;
        }
    }
    . . .
}
```

13.9. Получение сведений о текущем пользователе

Чтобы получить объект модели `User`, хранящий текущего пользователя, следует выполнить любое из следующих действий:

- вызвать у фасада `Illuminate\Support\Facades\Auth` метод `user()`:

```
<p>Добро пожаловать, {{ Auth:user()->name }}!</p>
```

- вызвать у объекта текущего клиентского запроса (как его получить, было показано в *разд. 9.3*) метод `user()`:

```
$currentUser = request()->user();
```

Получить ключ текущего пользователя можно вызовом у фасада `Auth` метода `id()`:

```
use Illuminate\Support\Facades\Auth;
. . .
$currentUserId = Auth::id();
```

Проверить, был ли выполнен вход, можно вызовом у фасада `Auth` метода `check()`. Если вход был выполнен, метод вернет `true`, в противном случае — `false`. Пример:

```
if (Auth::check()) {
    // Вход был выполнен
} else {
    // Вход не был выполнен
}
```

При вызове методов непосредственно у фасада `Auth` задействуется страж по умолчанию. Если нужно указать другой страж, следует использовать метод `guard()`, описанный в *разд. 13.5*. Пример:

```
$currentUser = Auth::guard('my_guard')->user();
```

Вместо фасада `Auth` можно использовать функцию `auth([<имя стража>=null])`. Если *имя стража* не указано, будет задействован страж по умолчанию. Примеры:

```
<p>Добро пожаловать, {{ auth()->user()->name }}!</p>
. . .
$currentUser = auth('my_guard')->user();
```

13.10. Подтверждение пароля

Перед тем как дать доступ к особо конфиденциальной информации (например, сведениям о самом пользователе), имеет смысл удостовериться, является ли текущий пользователь тем, за кого он себя выдает. Удостовериться в этом проще всего, попросив пользователя повторно ввести его пароль.

Чтобы запросить подтверждение пароля перед переходом на какую-либо страницу, достаточно связать указывающий на него маршрут с посредником `password.confirm`:

```
Route::get('/home', [HomeController::class, 'profile'])
    ->middleware('password.confirm');
```

Также на страницу подтверждения пароля пользователь может попасть непосредственно — перейдя по пути, по которому она находится (изначально — **/password/confirm**).

Подтверждение пароля реализует контроллер `Auth\ConfirmPasswordController`. Всю функциональность он получает из трейта `Illuminate\Foundation\Auth\ConfirmsPasswords` (хранится в модуле vendor\laravel\ui\auth-backend\ConfirmsPasswords.php).

В этом трейте объявлены оба действия контроллера:

- `showConfirmForm()` — выводит страницу подтверждения на основе шаблона auth\passwords\confirm.blade.php. Изначально веб-форма подтверждения содержит поле ввода для занесения пароля, кнопку отправки данных и гиперссылку на страницу сброса пароля;
- `confirm()` — выполняет подтверждение. Сначала проверяет занесенные в веб-форму данные на корректность и на совпадение введенного пароля с хранящимся в списке пользователей. Если пароли совпадают, сохраняет текущую временну́ю отметку в сессии, чтобы впоследствии проверить, не устарел ли подтвержденный пароль, и, если посетитель попал на страницу подтверждения пароля:
  - непосредственно — вызывает метод `redirectPath()` (был описан в *разд. 13.5*) для получения адреса перенаправления;
  - опосредованно — выполняет перенаправление на страницу, на которую пытался попасть пользователь.

  Если пароли не совпадают, действие повторно выводит страницу подтверждения с соответствующим сообщением.

В конструкторе контроллера `Auth\ConfirmPasswordController` выполняется связывание всех его действий с посредником `auth`.

Изменить функциональность контроллера можно, переопределив в нем следующие свойства и методы:

- `rules()` — метод, защищенный, объявлен в трейте `ConfirmsPassword`. Должен возвращать массив с наборами правил валидации для данных, введенных в веб-форму проверки пароля. Изначально возвращает массив с единственным набором правил — для проверки пароля, который должен быть указан и должен совпадать с паролем текущего пользователя;
- `validationErrorMessages()` — метод, защищенный, объявлен в трейте `ConfirmsPassword`. Должен возвращать массив с сообщениями об ошибках ввода. Изначально возвращает «пустой» массив;

- `redirectTo` — свойство, аналогично одноименному из контроллера `Auth\RegisterController` (см. *разд. 13.5*);
- `redirectTo()` — метод, описан в *разд. 13.5*.

13.11. Выход с веб-сайта

Выход с сайта реализован в действии `logout()` контроллера `Auth\LoginController`, унаследованном от трейта `AuthenticatesUsers`. Сначала оно собственно выполняет выход, очищает серверную сессию и генерирует для нее новый идентификатор. Далее пытается получить серверный ответ, сообщающий об успешном выходе (страницу с соответствующим сообщением или перенаправление на эту страницу), вызвав метод `loggedOut()`, и в случае успеха пересылает полученный ответ клиенту. Если метод `loggedOut()` «пуст» (это его изначальное состояние), выполняет перенаправление в «корень» сайта.

Защищенный метод `loggedOut(Request $request)` объявлен в трейте `AuthenticatesUsers`. С параметром `request` он принимает объект текущего клиентского запроса. Должен возвращать серверный ответ после успешного выхода: какую-либо страницу (например, с сообщением об успешном выходе) или перенаправление по произвольному адресу (например, на страницу с таким сообщением). Изначально этот метод «пуст».

13.12. Проверка существования адреса электронной почты

Многие сайты после регистрации нового пользователя проверяют, существует ли указанный им при регистрации адрес электронной почты. На этот адрес отсылается письмо с гиперссылкой, по которой новый пользователь должен выполнить переход и тем самым подтвердить существование адреса.

Чтобы реализовать такую проверку в Laravel-проекте, сначала необходимо настроить подсистему электронной почты. Как это сделать, будет рассказано в *главе 23*.

Далее следует реализовать в классе модели пользователя `User` интерфейс `Illuminate\Contracts\Auth\MustVerifyEmail` (помечающий модель как способную проверять адрес) и присоединить к этому классу трейт `Illuminate\Auth\MustVerifyEmail` (содержащий ряд необходимых методов). Изначально интерфейс в модели `User` не реализован (хотя и импортирован), а трейт даже не импортирован. Вот необходимый код:

```
use Illuminate\Auth\MustVerifyEmail as VerifyEmail;
class User extends Authenticatable implements MustVerifyEmail {
    use VerifyEmail;
    . . .
}
```

По умолчанию маршруты, ведущие на контроллер, который реализует проверку адреса, не создаются. Чтобы они создавались, следует явно указать это в вызове метода `routes()` фасада `Auth` (подробности — в *разд. 13.3*):

```
Auth::routes(['verify' => true]);
```

Указать, что на какую-либо страницу может попасть лишь пользователь с подтвержденным адресом, можно, связав ведущий на эту страницу маршрут с посредником `verified`:

```
Route::get('/home', [HomeController::class, 'profile'])
    ->middleware('verified');
```

Также на страницу проверки адреса пользователь может попасть непосредственно — перейдя по пути, по которому она находится (изначально — **/email/verify**).

Если пользователь шесть раз подряд запустил отправку электронного письма с гиперссылкой, страница проверки будет заблокирована на минуту.

Проверку адреса проводит контроллер `Auth\VerificationController`. Всю функциональность он получает из трейта `Illuminate\Foundation\Auth\VerifiesEmails` (хранится в модуле vendor\laravel\ui\auth-backend\VerifiesEmails.php).

В этом трейте объявлены три действия контроллера:

- `show()` — выводит страницу проверки. Сначала выясняет, не был ли почтовый адрес проверен ранее, и, если это так, вызывает метод `redirectPath()` (был описан в *разд. 13.5*). В противном случае генерирует страницу на основе шаблона auth\verify.blade.php. Страница изначально содержит только веб-форму с кнопкой, запускающей отправку письма;
- `resend()` — отправляет письмо. Сначала проверяет, не был ли адрес проверен ранее, и, если так, в зависимости от того, как пользователь попал на страницу:
  - непосредственно — вызывает метод `redirectPath()` (был описан в *разд. 13.5*), чтобы получить интернет-адрес для перенаправления;
  - опосредованно — выполняет перенаправление на страницу, на которую пытался попасть посетитель.

  Если адрес еще не проверялся, отсылает письмо вызовом метода `sendEmailVerificationNotification()` класса модели пользователя и снова выводит страницу проверки адреса с соответствующим сообщением;
- `verify()` — помечает адрес как проверенный. Сначала проверяет совпадение ключа пользователя, извлеченного из интернет-адреса, с сохраненным в списке пользователей, а также подлинность цифровой подписи, которой был подписан интернет-адрес, и если ключ и цифровая подпись не прошли проверку, возбуждает исключение `AuthorizationException`. Далее выясняет, не был ли почтовый адрес проверен ранее, и в таком случае выполняет перенаправление, аналогичное проводимому действием `resend()`.

  Подтверждение адреса электронной почты действие выполняет, занося в поле `email_verified_at` пользователя текущую временну́ю отметку, затем генерирует событие `Verified`. Далее, если посетитель попал на страницу входа:
  - непосредственно — пытается получить серверный ответ, сообщающий об успешной проверке (страницу с соответствующим сообщением или перенаправление на эту страницу), вызвав метод `verified()`, и в случае успеха пересылает полученный ответ клиенту. Если метод `verified()` «пуст» (его изначальное состояние), вызывает метод `redirectPath()` (был описан в *разд. 13.5*);

- опосредованно — выполняет перенаправление на страницу, на которую пытался попасть посетитель.

В конструкторе контроллера выполняется связывание со следующими посредниками:

- ❒ всех действий — с посредником `auth`;
- ❒ действия `verify` — с посредником `signed` (требует, чтобы интернет-адрес был подписан);
- ❒ действий `verify` и `resend` — с посредником `throttle:6,1` (после шести отправок письма заблокировать действие на одну минуту).

По умолчанию отправляемое письмо формируется на основе класса оповещения `Illuminate\Auth\Notifications\VerifyEmail`, входящего в состав фреймворка (оповещения будут рассмотрены в *главе 24*). Если стандартное содержание этого оповещения по какой-либо причине нас не устраивает, мы можем изменить его. Для этого следует в теле метода `boot()` провайдера `App\Providers\AuthServiceProvider` записать вызов статического метода `toMailUsing(<`*`анонимная функция`*`>)` класса `VerifyMail`. Задаваемая *анонимная функция* должна принимать в качестве параметров объект пользователя, выполняющего подтверждение адреса электронной почты, и строку с интернет-адресом, на который требуется выполнить переход для такого подтверждения. *Функция* должна создавать объект класса `MailMessage`, заносить него необходимое содержимое, пользуясь соответствующими программными инструментами (см. *разд. 24.1.1*), и возвращать этот объект. Пример:

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        VerifyEmail::toMailUsing(function ($user, $url) {
            return (new MailMessage)
                  ->subject('Проверка адреса электронной почты')
                  ->line('Уважаемый ' . $user->name . '!')
                  ->line('Нажмите кнопку внизу.')
                  ->action('Проверить адрес', $url);
        });
    }
}
```

Также можно указать свое оповещение, переопределив в модели пользователя общедоступный метод `sendEmailVerificationNotification()` (наследуется из трейта `MustVerifyEmail`). Пример кода переопределенного метода, генерирующего письмо на основе оповещения `MyVerifyEmail`:

```
class User extends Authenticatable {
    . . .
    public function sendEmailVerificationNotification() {
        $this->notify(new MyVerifyEmail);
    }
}
```

Метод `notify()` модели отправляет оповещение, чей объект указан в параметре.

Изменить функциональность контроллера можно, переопределив в нем следующие свойства и методы:

- `redirectTo` — свойство, аналогичное одноименному из контроллера `Auth\RegisterController` (см. *разд. 13.5*);
- `redirectTo()` — метод, описан в *разд. 13.5*;
- `verified(Request $request)` — метод, защищенный, объявлен в трейте `VerifiesEmails`. С параметром `request` принимает объект текущего клиентского запроса. Должен возвращать серверный ответ после успешной проверки: какую-либо страницу (например, с сообщением об успешной проверке) или перенаправление по произвольному адресу (например, на страницу с таким сообщением). Изначально «пуст».

> ***Попытка проверить адрес пользователя на существование…***
>
> …предпринимается сразу же после регистрации этого пользователя. Чтобы соответствующее письмо было успешно отправлено, в классе модели `User` следует реализовать интерфейс `MustVerifyEmail` и включить одноименный трейт.

13.13. Сброс пароля

Для реализации сброса пароля в Laravel-проекте сначала следует реализовать в классе модели пользователя `User` интерфейс `Illuminate\Contracts\Auth\CanResetPassword` (помечающий модель как способную выполнять сброс пароля) и присоединить к этому классу трейт `Illuminate\Auth\Passwords\CanResetPassword` (содержащий необходимые методы). Изначально интерфейс и трейт там даже не импортированы. Вот необходимый код:

```
use Illuminate\Contracts\Auth\CanResetPassword;
use Illuminate\Auth\Passwords\CanResetPassword as ResetPassword;
class User extends Authenticatable implements CanResetPassword {
    use ResetPassword;
    . . .
}
```

13.13.1. Отправка электронного письма с гиперссылкой сброса пароля

Отправку письма с гиперссылкой, ведущей на страницу сброса пароля, выполняет контроллер `Auth\ForgotPasswordController`. Всю функциональность он получает от трейта `Illuminate\Foundation\Auth\SendsPasswordResetEmails` (хранится в модуле vendor\laravel\ui\auth-backend\SendsPasswordResetEmails.php).

В трейте объявлены оба действия контроллера:

- `showLinkRequestForm()` — выводит страницу отправки письма с гиперссылкой. Страница генерируется шаблоном auth\passwords\email.blade.php и содержит веб-форму с полем ввода для занесения адреса электронной почты, по которому будет отправлено письмо, и кнопку отправки данных;
- `sendResetLinkEmail()` — собственно отправляет письмо.

По умолчанию отправляемое письмо формируется на основе класса оповещения `Illuminate\Auth\Notifications\ResetPassword`, входящего в состав фреймворка и принимающего в качестве параметра электронный жетон сброса пароля. Чтобы изменить содержание этого оповещения, достаточно в теле метода `boot()` провайдера `App\Providers\AuthServiceProvider` записать вызов статического метода `toMailUsing(<анонимная функция>)` класса `ResetPassword`. Задаваемая *анонимная функция* должна принимать в качестве параметров объект пользователя, выполняющего сброс пароля, и строку с жетоном сброса пароля, создавать объект класса `MailMessage`, заносить него необходимое содержимое и возвращать этот объект. Пример:

```
use Illuminate\Auth\Notifications\ResetPassword;
use Illuminate\Notifications\Messages\MailMessage;
class AuthServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        ResetPassword::toMailUsing(function ($user, $token) {
            $url = route('password.reset', [
                'token' => $token,
                'email' => $user->email,
            ]);
            return (new MailMessage)
                  ->subject('Сброс пароля')
                  ->line('Уважаемый ' . $user->name . '!')
                  ->line('Нажмите кнопку внизу.')
                  ->action('Сбросить пароль', $url);
        });
    }
}
```

Также можно использовать свое собственное оповещение, переопределив в классе модели пользователя общедоступный метод `sendPasswordResetNotification()` (наследуется от трейта `CanResetPassword`). Пример кода переопределенного метода, генерирующего письмо на основе оповещения `MyResetPasswordEmail`:

```
class User extends Authenticatable {
    . . .
    public function sendPasswordResetNotification($token) {
        $this->notify(new MyResetPasswordEmail($token));
    }
}
```

13.13.2. Собственно сброс пароля

Сбросом пароля занимается контроллер `Auth\ResetPasswordController`. Всю функциональность он получает из трейта `Illuminate\Foundation\Auth\ResetsPasswords` (хранится в модуле vendor\laravel\ui\auth-backend\ResetsPasswords.php).

В трейте объявлены оба действия контроллера:

- `showResetForm()` — выводит страницу сброса пароля. Страница генерируется шаблоном auth\passwords\reset.blade.php, которому в составе контекста передаются пере-

менные `token` (электронный жетон сброса пароля) и `email` (адрес электронной почты пользователя).

Изначально веб-форма сброса содержит скрытое поле с электронным жетоном сброса (берется из переменной `token`), поля ввода для занесения адреса электронной почты (из переменной `email`), пароля, его подтверждения и кнопку отправки данных.

- `reset()` — собственно сбрасывает пароль. Сначала проверяет корректность занесенных данных, записывает новый пароль, генерирует событие `PasswordReset`, выполняет вход и производит перенаправление по адресу, возвращенному методом `redirectPath()` (был описан в *разд. 13.5*). Если занесенный в веб-форму адрес электронной почты не совпадает с записанным в списке пользователей, вновь выводит страницу сброса пароля с соответствующим сообщением.

Изменить функциональность контроллера можно, переопределив в нем следующие свойства и методы:

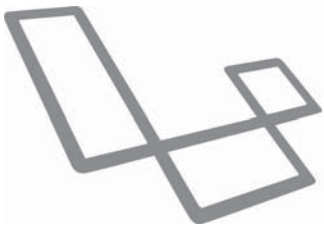
- `rules()` — метод, защищенный, объявлен в трейте `ResetsPasswords`. Должен возвращать ассоциативный массив с правилами валидации. Изначально возвращает массив со следующими правилами:
  - `token` (электронный жетон сброса пароля) — обязателен для указания;
  - `email` — обязателен для указания, адрес электронной почты;
  - `password` — обязателен для указания, минимальная длина — 8 символов, должен быть подтвержден в элементе управления `password_confirmed`;
- `validationErrorMessages()` — метод, защищенный, объявлен в трейте `ResetsPasswords`. Должен возвращать ассоциативный массив с сообщениями об ошибках. Изначально возвращает «пустой» массив;
- `redirectTo` — свойство, аналогичное одноименному из контроллера `Auth\RegisterController` (см. *разд. 13.5*);
- `redirectTo()` — метод, описан в *разд. 13.5*;
- `guard()` — метод, аналогичен одноименному из трейта `RegistersUsers` (см. *разд. 13.5*).

13.13.3. Удаление устаревших жетонов сброса пароля

Электронный жетон сброса пароля действителен в течение определенного времени, указанного в рабочей настройке `auth.passwords.expire`. По истечении этого времени он становится неактуальным, однако продолжает храниться в таблице жетонов. На сайте с большим количеством пользователей устаревших жетонов может накопиться довольно много.

Для удаления устаревших жетонов сброса пароля следует использовать команду:

```
php artisan auth:clear-resets
```



ГЛАВА 14

Обработка строк, массивов и функции-хелперы

Laravel предоставляет развитые инструменты для обработки строк и массивов, а также функции-хелперы.

> ***Полезно знать...***
>
> Инструменты Laravel, обрабатывающие строки и массивы, представляют собой удобные объектные «обертки» инструментов, встроенных в PHP.

14.1. Обработка строк

Средства для обработки строк, предоставляемые фреймворком, делятся на две части:

- статические методы класса `Illuminate\Support\Str`:

```
>>> // Получаем длину строки
>>> use Illuminate\Support\Str;
>>> echo Str::length('Фреймворк Laravel');
17
```

- методы класса `Illuminate\Support\Stringable`. Этот класс предназначен для хранения обычной строки PHP и предоставляет ряд удобных методов для ее обработки. При выводе и в любых операциях, требующих строкового аргумента, объект этого класса автоматически преобразуется в строку.

  Объект класса `Stringable` можно создать тремя способами:

  - вызовом конструктора в формате `Stringable(<`*строка*`>)`:

    ```
    >>> use \Illuminate\Support\Stringable;
    >>> $str = new Stringable('Фреймворк Laravel');
    ```

  - вызовом статического метода `of(<`*строка*`>)` класса `Str`:

    ```
    >>> $str = Str::of('Фреймворк Laravel');
    ```

  - вызвав функцию-хелпер `str(<`*строка*`>)`:

    ```
    >>> $str = str('Фреймворк Laravel');
    ```

Пример получения длины строки:

```
>>> echo $str->length();
17
```

Все без исключения методы класса `Stringable`, кроме выдающих в качестве результата логические величины (даже использованный в приведенном ранее примере `length()`), возвращают результат, также представленный объектом класса `Stringable`. Это позволяет записывать «сцепки» методов, наподобие:

```
>>> echo Str::of('Фреймворк')->append(' ')->append('Laravel')
...                                    ->length();
17
```

Многие методы классов `Str` и `Stringable` имеют одинаковые имена и выполняют одинаковые действия, поэтому далее они будут описываться совместно. Методы, не имеющие одноименной «пары», будут помечены.

> ***Все классы и функции, описываемые далее, поддерживают UTF-8***
> Поэтому их можно использовать для обработки строк на русском языке.

14.1.1. Составление строк

- `append(<строка>)` (только класс `Stringable`) — добавляет заданную *строку* в конце текущей:

  ```
  >>> echo Str::of('Фреймворк')->append(' ')->append('Laravel');
  Фреймворк Laravel
  ```

- `prepend(<строка>)` (только класс `Stringable`) — добавляет заданную *строку* в начале текущей:

  ```
  >>> echo Str::of('Laravel')->prepend(' ')->prepend('Фреймворк');
  Фреймворк Laravel
  ```

- `newline()` (только класс `Stringable`) — добавляет к концу текущей строки символ разрыва строки:

  ```
  >>> echo str('Laravel')->newLine()->append('Введение');
  Laravel
  Введение
  ```

- `start()` — добавляет к *строке* заданный *префикс*, только если он там отсутствует:

  ```
  Str::start(<строка>, <префикс>)
  <строка Stringable>->start(<префикс>)
  ```

  Примеры:

  ```
  >>> echo Str::of('background.jpg')->start('/');
  /background.jpg
  >>> echo Str::of('/background.jpg')->start('/');
  /background.jpg
  ```

- `finish()` — добавляет к *строке* заданный *постфикс*, только если он там отсутствует:

  ```
  Str::finish(<строка>, <постфикс>)
  <строка Stringable>->finish(<постфикс>)
  ```

Примеры:

```
>>> echo Str::of('background')->finish('.jpg');
background.jpg
>>> echo Str::of('background.jpg')->finish('.jpg');
background.jpg
```

14.1.2. Сравнение строк и получение сведений о строках

- `exactly(<строка>)` (только класс `Stringable`) — возвращает `true`, если заданная *строка* полностью совпадает с текущей, и `false` — в противном случае:

```
>>> echo Str::of('12345678')->exactly('12345678');
true
>>> echo Str::of('12345678')->exactly(12345678);
false
```

- `is()` — возвращает `true`, если заданная *строка* совпадает с указанным *шаблоном*, и `false` — в противном случае. В шаблоне можно использовать литерал `*` для обозначения произвольного количества любых символов. Формат вызова:

```
Str::is(<строка>, <шаблон>)
<строка Stringable>->is(<шаблон>)
```

Примеры:

```
>>> $str = Str::of('BbController');
>>> echo $str->is('*Controller');
true
>>> echo $str->is('*Policy');
false
```

- `length()` — возвращает длину строки в символах:

```
Str::length(<строка>)
<строка Stringable>->length()
```

- `wordCount()` — возвращает количество слов в строке. Формат вызова такой же, как и у метода `length()`. Пример:

```
>>> echo Str::wordCount('Laravel is the best!');
4
```

К сожалению, в случае строк, содержащих символы кириллицы, работает некорректно:

```
>>> echo Str::wordCount('Фреймворк Laravel лучше всех!');
1
```

- `isEmpty()` (только класс `Stringable`) — возвращает `true`, если текущая строка «пуста», и `false` — в противном случае:

```
>>> echo Str::of('')->isEmpty();
true
```

```
>>> echo Str::of('PHP')->isEmpty();
false
>>> echo Str::of('   ')->isEmpty();
false
```

- `isNotEmpty()` (только класс `Stringable`) — возвращает `true`, если текущая строка, наоборот, не «пуста», и `false` — в противном случае;
- `isAscii()` — возвращает `true`, если заданная *строка* содержит только символы из кодировки ASCII, и `false` — в противном случае:

  ```
  Str::isAscii(<строка>)
  <строка Stringable>->isAscii()
  ```

  Пример:

  ```
  >>> echo Str::isAscii('Framework');
  true
  >>> echo Str::isAscii('Фреймворк');
  false
  ```

- `isUuid(<строка>)` (только класс `Str`) — возвращает `true`, если переданная *строка* является универсальным уникальным идентификатором, и `false` — в противном случае:

  ```
  >>> echo Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');
  true
  >>> echo Str::isUuid('987654321');
  false
  ```

14.1.3. Преобразование строк

- `lower()` — возвращает *строку*, преобразованную к нижнему регистру:

  ```
  Str::lower(<строка>)
  <строка Stringable>->lower()
  ```

  Пример:

  ```
  >>> echo Str::lower('ФрЕйМвОрК');
  фреймворк
  ```

- `upper()` — возвращает *строку*, преобразованную к верхнему регистру. Формат вызова такой же, как и у метода `lower()`. Пример:

  ```
  >>> echo Str::upper('ФрЕйМвОрК');
  ФРЕЙМВОРК
  ```

- `ucfirst()` — возвращает *строку*, набранную с прописной буквы. Формат вызова такой же, как и у метода `lower()`. Пример:

  ```
  >>> echo Str::ucfirst('фреймворк');
  Фреймворк
  ```

- `lcfirst()` — возвращает *строку*, набранную со строчной буквы. Формат вызова такой же, как и у метода `lower()`. Пример:

```
>>> echo Str::lcfirst('Фреймворк');
фреймворк
```

- `title()` — возвращает *строку*, каждое слово которой набрано с прописной буквы. Формат вызова такой же, как и у метода `lower()`. Пример:

```
>>> echo Str::title('фреймворк laravel');
Фреймворк Laravel
```

- `limit()` — возвращает *строку*, обрезанную до заданной длины. Можно указать *завершающий постфикс*, которым будет заканчиваться обрезанная строка (по умолчанию: `'...'`). Формат вызова:

```
Str::limit(<строка>, <длина>[, <завершающий постфикс>='...'])
<строка Stringable>->limit(<длина>[, <завершающий постфикс>='...'])
```

Примеры:

```
>>> $str = Str::of('Каждый охотник желает знать, где спрятался фазан');
>>> echo $str->limit(25);
Каждый охотник желает зна...
>>> echo $str->limit(25, ' ->');
Каждый охотник желает зна ->
```

- `words()` — возвращает *строку*, обрезанную до заданного *количества слов*. Можно указать *завершающий постфикс*, которым будет заканчиваться обрезанная строка (по умолчанию: три точки). Формат вызова:

```
Str::words(<строка>, [<количество слов>=100[,
                      <завершающий постфикс>='...']])
<строка Stringable>->words([<количество слов>=100[,
                           <завершающий постфикс>='...']])
```

Примеры:

```
>>> echo $str->words(5);
Каждый охотник желает знать, где...
>>> echo $str->words(5, '>>>');
Каждый охотник желает знать, где>>>
```

- `squish()` — удаляет из заданной *строки* повторяющиеся пробелы и возвращает результат:

```
Str::squish(<строка>)
<строка Stringable>->squish()
```

Пример:

```
>>> Str::squish('   Laravel   9       ');
Laravel 9
```

- `trim()` (только класс `Stringable`) — удаляет у текущей строки начальные и конечные пробелы и возвращает результат;
- `ltrim()` (только класс `Stringable`) — удаляет у текущей строки начальные пробелы и возвращает результат;
- `rtrim()` (только класс `Stringable`) — удаляет у текущей строки конечные пробелы и возвращает результат;

- `remove()` — удаляет из заданной *строки* указанный *символ* или *символы* и возвращает результат:

```
Str::remove(<удаляемый символ>|<массив с удаляемыми символами>,
            <строка>[, <с учетом регистра символов?>=true])
<строка Stringable>->remove(<удаляемый символ>|↳
                            <массив с удаляемыми символами>[,
                            <с учетом регистра символов?>=true])
```

  По умолчанию поиск *удаляемых символов* в *строке* производится с учетом регистра. Чтобы производился поиск без учета регистра, следует дать параметру с *учетом регистра* значение `false` (к сожалению, с кириллическими символами это не работает). Примеры:

```
>>> echo Str::remove('й', 'Фреймворк');
Фремворк
>>> echo Str::remove(['й', 'Ф'], 'Фреймворк', false);
ремворк
```

- `slug()` — возвращает слаг, созданный на основе указанной *строки*, в котором отдельные слова отделены друг от друга заданным *разделителем*:

```
Str::slug(<строка>[, <разделитель>='-'])
<строка Stringable>->slug([<разделитель>='-'])
```

  Примеры:

```
>>> echo Str::slug('Фреймворк Laravel');
freimvork-laravel
>>> echo Str::slug('Фреймворк Laravel', '_');
freimvork_laravel
```

- `ascii()` — возвращает *строку*, транслитерированную в символы кодировки ASCII:

```
Str::ascii(<строка в кодировке UTF-8>)
<строка в кодировке UTF-8 Stringable>->ascii()
```

  Пример:

```
>>> echo Str::of('Фреймворк')->ascii();
Freimvork
```

- `studly()` — возвращает *строку*, в которой все отдельные слова набраны слитно и с прописной буквы (стиль именования PascalCase):

```
Str::studly(<строка>)
<строка Stringable>->studly()
```

  Пример:

```
>>> echo Str::studly('machines_spares');
MachinesSpares
```

- `camel()` — возвращает *строку*, преобразованную согласно стилю именования camelCase. Формат вызова такой же, как и у метода `studly()`. Пример:

```
>>> echo Str::camel('machines_spares');
machinesSpares
```

- `kebab()` — возвращает *строку*, преобразованную согласно стилю kebab-case. Формат вызова такой же, как и у метода `studly()`. Пример:

```
>>> echo Str::kebab('machinesSpares');
machines-spares
```

- `snake()` — возвращает *строку*, преобразованную согласно стилю snake_case. Формат вызова такой же, как и у метода `studly()`. Пример:

```
>>> echo Str::snake('machinesSpares');
machines_spares
```

- `headline(<строка>)` (только класс `Str`) — возвращает указанную строку, преобразованную из стилей camelCase, kebab-case, snake_case или PascalCase в обычный стиль именования, в котором каждое слово начинается с прописной буквы:

```
>>> echo Str::headline('camelCase');
Camel Case
>>> echo Str::headline('kebab-case');
Kebab Case
>>> echo Str::headline('snake_case');
Snake Case
>>> echo Str::headline('PascalCase');
Pascal Case
```

- `plural()` — возвращает строку, в которой последнее существительное преобразовано во множественное число. Поддерживается только английский язык. Формат вызова такой же, как и у метода `studly()`. Примеры:

```
>>> echo Str::plural('page');
pages
>>> echo Str::plural('child');
children
>>> echo Str::plural('child good child');
child good children
```

- `pluralStudly(<строка>[, <количество>=2])` (только класс `Str`) — возвращает заданную в стиле PascalCase *строку* в единственном или множественном числе, в зависимости от указанного *количества*:

```
>>> echo Str::pluralStudly('MainController');
MainControllers
>>> echo Str::pluralStudly('MainController', 1);
MainController
```

- `singular()` — возвращает строку, в которой последнее существительное преобразовано в единственное число. Поддерживается только английский язык. Формат вызова такой же, как и у метода `studly()`. Примеры:

```
>>> echo Str::singular('pages');
page
>>> echo Str::singular('children');
child
>>> echo Str::singular('children good children');
children good child
```

- `padBoth()` — добавляет с обоих концов заданной *строки* указанные *символы*, чтобы строка получила заданную *длину*, и возвращает результат:

```
Str::padBoth(<строка>, <длина>[, <добавляемый символ>=' '])
<строка Stringable>->padBoth(<длина>[, <добавляемый символ>=' '])
```

  Примеры:

```
>>> echo Str::padBoth('Laravel', 20)
      Laravel
>>> echo Str::padBoth('Laravel', 20, '.')
......Laravel.......
>>> echo Str::padBoth('Laravel', 20, '. ')
... Laravel... .
```

- `padLeft()` — то же самое, что и `padBoth()`, только добавляет указанные символы лишь слева;
- `padRight()` — то же самое, что и `padBoth()`, только добавляет указанные символы лишь справа:

```
>>> echo Str::padRight('Laravel', 20, '.')
Laravel.............
```

- `markdown()` — преобразует заданную *строку* из формата Markdown в HTML и возвращает результат. Для преобразования используется библиотека CommonMark (**https://commonmark.thephpleague.com/**). Форматы вызова:

```
Str::markdown(<строка>[, <настройки преобразования>=[]])
<строка Stringable>->markdown([<настройки преобразования>=[]])
```

  Можно указать *настройки преобразования*, поддерживаемые библиотекой CommonMark, в виде ассоциативного массива. Ключи элементов этого массива должны соответствовать отдельным настройкам, а значения элементов зададут значения этих настроек. Пример:

```
>>> echo Str::markdown("# Laravel\r\n## Введение");
<h1>Laravel</h1>
<h2>Введение</h2>
```

- `pipe(<функция>)` (только класс `Stringable`) — передает текущую строку в качестве единственного параметра указанной *функции* и возвращает выданный ею результат в виде нового объекта класса `Stringable`. В параметре можно указать как имя обычной именованной функции, так и анонимную функцию. Примеры:

```
>>> echo str('Laravel')->pipe('md5');
a5c95b86291ea299fcbe64458ed12702
>>> echo str('Laravel')->pipe(function ($s) { return '"' . $s . '"'; });
"Laravel"
```

- `mask()` — заменяет указанную часть заданной *строки* указанным *символом*, тем самым маскируя ее, и возвращает результат:

```
Str::mask(<строка>, <маскирующий символ>,
          <индекс первого символа маскируемой части>[,
          <длина маскируемой части в символах>=null])
```

```
<строка Stringable>->mask(<маскирующий символ>,
                          <индекс первого символа маскируемой части>[,
                          <длина маскируемой части в символах>=null])
```

Нумерация символов в *строке* начинается с нуля. Если в качестве *индекса первого символа* указано отрицательное число, нумерация начнется с конца строки. Если не указана *длина маскируемой части*, замаскированы будут все оставшиеся символы строки. Примеры:

```
>>> echo str('admin#bboard.ru')->mask('*', 3);
adm************
>>> echo str('admin#bboard.ru')->mask('*', 3, 2);
adm**#bboard.ru
>>> echo str('admin#bboard.ru')->mask('*', -5, 2);
admin#bboa**.ru
```

- `reverse(<строка>)` (только класс `Str`) — возвращает заданную *строку*, у которой символы следуют друг за другом в обратном порядке:

```
>>> echo Str::reverse('Фреймворк Laravel');
levaraL кровмйерФ
```

14.1.4. Извлечение фрагментов строк

- `substr()` — возвращает фрагмент *строки*, начинающийся с символа с указанным *номером* и имеющий заданную *длину*. Нумерация символов в строке начинается с нуля. Если *длина* не задана, возвращенный фрагмент будет содержать все символы до конца *строки*. Формат вызова:

```
Str::substr(<строка>, <номер первого символа>[, <длина>=null])
<строка Stringable>->substr(<номер первого символа>[, <длина>=null])
```

Примеры:

```
>>> echo Str::substr('App\\Controller\\BbController', 4);
Controller\BbController
>>> echo Str::substr('App\\Controller\\BbController', 4, 10);
Controller
```

- `before()` — возвращает фрагмент *строки*, находящийся перед первым вхождением указанной *подстроки*:

```
Str::before(<строка>, <подстрока>)
<строка Stringable>->before(<подстрока>)
```

Если *подстрока* в *строке* отсутствует, возвращается сама *строка*. Пример:

```
>>> echo Str::before('App\\Http\\Controllers', '\\');
App
```

- `beforeLast()` — возвращает фрагмент *строки*, находящийся перед последним вхождением указанной *подстроки*. Формат вызова такой же, как и у метода `before()`. Если *подстрока* в *строке* отсутствует, возвращается сама *строка*. Пример:

```
>>> echo Str::beforeLast('App\\Http\\Controllers', '\\');
App\Http
```

- `after()` — возвращает фрагмент *строки*, находящийся после первого вхождения указанной *подстроки*. Формат вызова такой же, как и у метода `before()`. Если *подстрока* в *строке* отсутствует, возвращается сама *строка*. Пример:

```
>>> echo Str::after('App\\Http\\Controllers', '\\');
Http\Controllers
```

- `afterLast()` — возвращает фрагмент *строки*, находящийся после последнего вхождения указанной *подстроки*. Формат вызова такой же, как и у метода `before()`. Если *подстрока* в *строке* отсутствует, возвращается сама *строка*. Пример:

```
>>> echo Str::afterLast('App\\Http\\Controllers', '\\');
Controllers
```

- `between()` (только класс `Str`) — возвращает самый длинный фрагмент *строки*, находящийся между *подстроками 1* и *2*:

```
Str::between(<строка>, <подстрока 1>, <подстрока 2>)
```

  Пример:

```
>>> echo Str::between('App\\Http\\Controllers\\HomeController',
...                   '\\', '\\');
Http\Controllers
```

- `betweenFirst()` — то же самое, что и `between()`, только возвращает самый короткий фрагмент:

```
>>> echo Str::betweenFirst('App\\Http\\Controllers\\HomeController',
...                        '\\', '\\');
Http
```

- `excerpt()` — возвращает отрывок из заданной *строки*, содержащий указанный *фрагмент*:

```
Str::excerpt(<строка>, <фрагмент>[, <настройки>=[]])
<строка Stringable>->excerpt(<фрагмент>[, <настройки>=[]])
```

  Настройки указываются в виде ассоциативного массива, ключи элементов которого соответствуют отдельным настройкам. В настоящее время поддерживаются следующие настройки:

  - `radius` — размер выдаваемого отрывка в виде количества символов слева и справа от заданного *фрагмента* (по умолчанию: 100);
  - `omission` — подстрока, которая будет подставлена в начале и конце отрывка (по умолчанию: `'...'`).

  Примеры:

```
>>> $s = 'Laravel - это полнофункциональный веб-фреймворк, ↳
предназначенный для разработки сайтов как общего назначения, так и ↳
специализированных';
>>> echo Str::excerpt($s, 'веб', ['radius' => 30]);
...vel - это полнофункциональный веб-фреймворк, предназначенный дл...
>>> echo Str::excerpt($s, 'веб', ['radius' => 10]);
...иональный веб-фреймворк...
```

```
>>> echo Str::excerpt($s, 'веб',
...                   ['radius' => 10 , 'omission' => '(...)']);
(...)иональный веб-фреймворк(...)
```

- `ucsplit()` — разделяет заданную *строку* на отдельные слова по прописным буквам и возвращает массив с полученными словами:

  ```
  Str::ucsplit(<строка>)
  <строка Stringable>->ucfirst()
  ```

  Пример:

  ```
  >>> Str::ucsplit('HomeController');
  => [ "Home", "Controller" ]
  ```

- `explode()` (только класс `Stringable`) — разделяет текущую строку на части по заданному *разделителю* и возвращает коллекцию Laravel, содержащую все полученные части. Во втором параметре можно указать максимальное количество элементов, которые будут присутствовать в возвращаемой коллекции. Формат вызова:

  ```
  explode(<разделитель>[,
          <максимальное количество элементов>=PHP_INT_MAX])
  ```

  Примеры:

  ```
  >>> $str = Str::of('PHP MySQL Laravel Node Vue');
  >>> $coll = $str->explode(' ');
  => [ "PHP", "MySQL", "Laravel", "Node", "Vue" ]
  >>> $coll = $str->explode(' ', 3);
  => [ "PHP", "MySQL", "Laravel Node Vue" ]
  ```

- `split()` (только класс `Stringable`) — разделяет текущую строку на части по *разделителю*, заданному в виде регулярного выражения. Во втором параметре можно указать максимальное количество элементов, которые будут присутствовать в возвращаемой коллекции. Формат вызова:

  ```
  split(<разделитель>[, <максимальное количество элементов>=-1[,
                       <настройки>=0]])
  ```

  Указанные *настройки* будут переданы последним параметром в функцию PHP `preg_split()`, которая и используется для разделения строки. Пример:

  ```
  >>> str('Модель, контроллер, шаблон')->split('/[\s,]+/');
  => ["Модель", "контроллер", "шаблон" ]
  ```

14.1.5. Поиск и замена в строках

- `contains()` — возвращает `true`, если строка содержит заданную *подстроку* или любую подстроку из содержащихся в *массиве*, и `false` — в противном случае:

  ```
  Str::contains(<строка>, <подстрока>|<массив подстрок>)
  <строка Stringable>->contains(<подстрока>|<массив подстрок>)
  ```

  Сравнение строк производится с учетом регистра символов. Примеры:

  ```
  >>> echo Str::contains('PHP, MySQL, Laravel', 'PHP');
  true
  ```

```
>>> echo Str::contains('PHP, MySQL, Laravel', 'JavaScript');
false
>>> echo Str::contains('PHP, MySQL, Laravel', ['PHP', 'Python']);
true
>>> echo Str::contains('PHP, MySQL, Laravel', ['Python', 'Ruby']);
false
```

- `containsAll()` — возвращает `true`, если строка содержит все подстроки из имеющихся в *массиве*, и `false` — в противном случае. Формат вызова такой же, как и у метода `contains()`. Пример:

```
>>> echo Str::containsAll('PHP, MySQL, Laravel', ['PHP', 'Python']);
false
>>> echo Str::containsAll('PHP, MySQL, Laravel', ['PHP', 'MySQL']);
true
```

- `startsWith()` — возвращает `true`, если строка начинается заданной *подстрокой* или любой подстрокой из содержащихся в *массиве*, и `false` — в противном случае. Формат вызова такой же, как и у метода `contains()`. Примеры:

```
>>> echo Str::startsWith('PHP, MySQL, Laravel', 'PHP');
true
>>> echo Str::startsWith('PHP, MySQL, Laravel', ['PHP', 'Python']);
true
>>> echo Str::startsWith('PHP, MySQL, Laravel', 'JavaScript');
false
```

- `endsWith()` — возвращает `true`, если строка заканчивается заданной *подстрокой* или любой подстрокой из содержащихся в *массиве*, и `false` — в противном случае. Формат вызова такой же, как и у метода `contains()`. Примеры:

```
>>> echo Str::endsWith('PHP, MySQL, Laravel', 'Laravel');
true
>>> echo Str::endsWith('PHP, MySQL, Laravel', ['Laravel', 'Yii']);
true
>>> echo Str::endsWith('PHP, MySQL, Laravel', 'Yii');
false
```

- `substrCount()` (только класс `Str`) — возвращает количество вхождений заданной *подстроки* в указанной *строке* или ее фрагменте указанной *длины*, начинающегося с символа с заданным *индексом*:

```
substrCount(<строка>, <подстрока>[,
                      <индекс первого символа фрагмента>=0[,
                      <длина фрагмента в символах>=null]])
```

Если *длина фрагмента* не указана, подсчет вхождений *подстроки* будет выполняться до конца *строки*. Пример:

```
>>> echo Str::substrCount('Laravel is great!', 'a');
3
>>> echo Str::substrCount('Laravel is great!', 'a', 2);
2
>>> echo Str::substrCount('Laravel is great!', 'a', 2, 4);
1
```

- `scan(<шаблон>)` (только класс `Stringable`) — разбирает текущую *строку* соответственно заданному *шаблону* и возвращает коллекцию `Collection` с результатом. Шаблон пишется в том же формате, что и у функции PHP `sscanf()`. Пример:

```
>>> $col = str('image.jpg')->scan('%[^.].%s');
>>> echo $col[0], ' ', $col[1];
image jpg
```

- `test(<регулярное выражение>)` (только класс `Stringable`) — возвращает `true`, если текущая строка совпадает с заданным *регулярным выражением*, и `false` — в противном случае:

```
>>> str('BbController')->test('/.*Controller/');
=> true
>>> str('BbPolicy')->test('/.*Controller/');
=> false
```

- `match(<регулярное выражение>)` (только класс `Stringable`) — возвращает первый фрагмент текущей строки, совпавший с заданным *регулярным выражением*. Если совпадение не было найдено, возвращается «пустая» строка. Примеры:

```
>>> echo Str::of('Фреймворк Laravel')->match('/[A-Za-z]+/');
Laravel
>>> echo Str::of('Framework Laravel')->match('/[A-Za-z]+/');
Framework
```

- `matchAll(<регулярное выражение>)` (только класс `Stringable`) — возвращает коллекцию всех фрагментов текущей строки, совпавших с заданным *регулярным выражением*. Если ни одно совпадение не было найдено, возвращается «пустая» коллекция. Пример:

```
>>> $coll = Str::of('Framework Laravel')->matchAll('/[A-Za-z]+/');
=> [ "Framework", "Laravel" ]
```

  Если в заданном *регулярном выражении* присутствует группа, возвращаемая коллекция будет содержать фрагменты, совпавшие с этой группой. Пример:

```
>>> $coll = Str::of('Framework Laravel')
...            ->matchAll('/[A-Z]([a-z]{4})/');
=> [ "rame", "arav" ]
```

- `substrReplace()` (только класс `Str`) — заменяет в заданной строке фрагмент указанной *длины*, начинающийся с символа с заданным *индексом*, на указанную *подстроку*:

```
substrReplace(<строка>, <подстрока>[,
                        <индекс первого символа фрагмента>=0[,
                        <длина фрагмента в символах>=null]])
```

  Если в качестве *длины фрагмента* указать 0, *подстрока* будет вставлена в позицию *строки*, имеющую указанный *индекс*. Пример:

```
>>> echo Str::substrReplace('GenericController', 'Home', 0, 7);
HomeController
>>> echo Str::substrReplace('GenericController', 'Home', 3, 4);
GenHomeController
```

```
>>> echo Str::substrReplace('GenericController', 'Home', 7, 0);
GenericHomeController
```

- replace(<*заменяемая подстрока*>, <*заменяющая подстрока*>) (только класс Stringable) — заменяет в текущей строке все вхождения *заменяемой подстроки заменяющей подстрокой* и возвращает результат:

```
>>> echo Str::of('- - -')->replace('-', '+');
+ + +
```

- replaceFirst() — заменяет в *строке* первое вхождение *заменяемой подстроки заменяющей подстрокой* и возвращает результат:

```
Str::replaceFirst(<заменяемая подстрока>, <заменяющая подстрока>, <строка>)
<строка Stringable>->replaceFirst(<заменяемая подстрока>,
                                  <заменяющая подстрока>)
```

Пример:

```
>>> echo Str::of('- - -')->replaceFirst('-', '+');
+ - -
```

- replaceLast() — заменяет в *строке* последнее вхождение *заменяемой подстроки заменяющей подстрокой* и возвращает результат. Формат вызова такой же, как и у метода replaceFirst(). Пример:

```
>>> echo Str::of('- - -')->replaceLast('-', '+');
- - +
```

- replaceArray() — заменяет в *строке* очередное вхождение *заменяемой подстроки* очередным элементом заданного *массива* и возвращает результат:

```
Str::replaceArray(<заменяемая подстрока>, <массив замен>, <строка>)
<строка Stringable>->replaceArray(<заменяемая подстрока>, <массив замен>)
```

Пример:

```
>>> echo Str::of('- - -')->replaceArray('-', ['+', '/', '\\']);
+ / \
```

Если в *строке* присутствует больше вхождений *заменяемой подстроки*, чем элементов в *массиве*, оставшиеся вхождения останутся неизменными. Пример:

```
>>> echo Str::of('- - -')->replaceArray('-', ['+', '/']);
+ / -
```

- replaceMatches() (только класс Stringable) — заменяет в текущей строке указанное *количество* фрагментов, совпадающих с заданным *регулярным выражением*, *заменяющими подстроками*. Если в качестве *количества* задано число -1, будут заменены все фрагменты. Формат вызова:

```
replaceMatches(<регулярное выражение>,
               <заменяющая подстрока>|<анонимная функция> [,
               <количество заменяемых фрагментов>=-1])
```

Примеры:

```
>>> $str = Str::of('1 500,473,240.34');
>>> echo $str->replaceMatches('/[, ]/', '');
1500473240.34
```

```
>>> echo $str->replaceMatches('/[,\\s]/', '', 2);
1500473,240.34
```

Вместо *заменяющей подстроки* можно задать *анонимную функцию*, принимающую массив, первым элементом которого будет очередной фрагмент, совпавший с *регулярным выражением*, вторым — фрагмент, совпавший с первой группой из присутствующих в *регулярном выражении*, третьим — фрагмент, совпавший со второй группой, и т. д. Возвращать эта *функция* должна подстроку, которой будет заменен очередной совпавший фрагмент. Пример:

```
>>> echo Str::of('html, css и php')
...         ->replaceMatches('/[a-z]+/', function ($match) {
...             return Str::upper($match[0]);
...         });
HTML, CSS и PHP
```

- `swap()` — выполняет в заданной *строке* замены, приведенные в указанном *массиве*, и возвращает результат:

```
Str::swap(<массив замен>, <строка>)
<строка Stringable>->swap(<массив замен>)
```

В *массиве замен* каждый элемент представляет одну замену, его ключ задает заменяемую подстроку, а значение — заменяющую. Пример:

```
>>> echo str('Django написан на языке Python')
...    ->swap(['Django' => 'Laravel', 'Python' => 'PHP']);
Laravel написан на языке PHP
```

Функция-хелпер `preg_replace_array()` заменяет в *строке* очередной фрагмент, совпавший с заданным *регулярным выражением*, очередным элементом *массива*, и возвращает результат:

```
preg_replace_array(<регулярное выражение>, <массив замен>, <строка>)
```

Пример:

```
>>> echo preg_replace_array('/:[a-z]+/', [100, 500],
...                         'Объявления с ценами от :start до :end');
Объявления с ценами от 100 до 500
```

14.1.6. Обработка путей к файлам

- `basename([<расширение>])` (только класс `Stringable`) — возвращает последний фрагмент пути (обычно это имя файла). Если задано *расширение*, оно будет удалено из возвращаемого фрагмента. Примеры:

```
>>> $str = Str::of('c:/sites/bboard/public/images/background.jpg');
>>> echo $str->basename();
background.jpg
>>> echo $str->basename('.jpg');
background
```

- `dirname([<уровень>=1])` (только класс `Stringable`) — возвращает путь без последних фрагментов, количество которых задано в параметре *уровень*:

```
>>> $str = Str::of('c:/sites/bboard/public/images/background.jpg');
>>> echo $str->dirname();
c:/sites/bboard/public/images
>>> echo $str->dirname(2);
c:/sites/bboard/public
```

14.1.7. Прочие инструменты для обработки строк

- `uuid()` (только класс `Str`) — генерирует и возвращает универсальный уникальный идентификатор версии 4;
- `orderedUuid()` (только класс `Str`) — генерирует и возвращает универсальный уникальный идентификатор, содержащий в своем начале текущую временну́ю отметку. Такой идентификатор используется в качестве ключа записи и при сортировке автоматически упорядочивает записи в порядке их создания;
- `random(<`*длина*`>)` (только класс `Str`) — генерирует и возвращает случайную строку заданной *длины*;
- `when()` (только класс `Stringable`) — если заданное *условие* истинно, вызывает *анонимную функцию 1*, передав ей в качестве параметра текущую строку, и возвращает выданный этой функцией результат. Если *условие* ложно, вызывает *анонимную функцию 2*, также передав ей в качестве параметра текущую строку, и также возвращает выданный этой функцией результат. Если *анонимная функция 2* не указана, ничего не делает. Формат вызова:

  ```
  when(<условие>, <анонимная функция 1>[, <анонимная функция 2>=null])
  ```

  Пример:

  ```
  $source = str($s);
  $result = when($source->length() > 20,
                 function ($s) { return $s->limit(20); },
                 function ($s) { return $s->padRight(20); }
                );
  ```

  В качестве *условия* также можно указать анонимную функцию. При вызове в качестве единственного параметра ей будет передана текущая строка, а возвращенный ею результат использован для работы. Пример:

  ```
  $result = when(function ($s) { return $s->length() > 20; }, ... );
  ```

- `unless()` (только класс `Stringable`) — аналогичен `when()`, только вызывает *анонимную функцию 1*, наоборот, если *условие* ложно, а *анонимную функцию 2* — если оно истинно;
- `whenExactly()` (только класс `Stringable`) — если текущая строка точно совпадает с заданной *подстрокой*, вызывает *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова:

  ```
  whenExactly(<подстрока>, <анонимная функция 1>[,
                           <анонимная функция 2>=null])
  ```

- `whenContains()` (только класс `Stringable`) — если текущая строка содержит заданную *подстроку* или любую подстроку из содержащихся в *массиве*, вызывает *анонимную*

функцию 1, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова:

```
whenContains(<подстрока>|<массив подстрок>,
             <анонимная функция 1>[, <анонимная функция 2>=null])
```

- `whenContainsAll()` (только класс `Stringable`) — аналогичен `whenContains()`, только текущая строка должна содержать все подстроки из приведенных в заданном *массиве*;
- `whenStartsWith()` (только класс `Stringable`) — если текущая строка начинается с заданной *подстроки* или любой подстроки из содержащихся в *массиве*, вызывает *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова такой же, как и у метода `whenContains()`;
- `whenEndsWith()` (только класс `Stringable`) — если текущая строка заканчивается заданной *подстрокой* или любой подстрокой из содержащихся в *массиве*, вызывает *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова такой же, как и у метода `whenContains()`;
- `whenIs()` (только класс `Stringable`) — если текущая строка совпадает с заданным *шаблоном*, вызывает *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. В *шаблоне* можно использовать литерал `*`, обозначающий произвольное количество любых символов. Формат вызова:

  ```
  whenIs(<шаблон>, <анонимная функция 1>[, <анонимная функция 2>=null])
  ```

- `whenTest()` (только класс `Stringable`) — если текущая строка совпадает с заданным *регулярным выражением*, вызывает *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова:

  ```
  whenTest(<регулярное выражение>, <анонимная функция 1>[,
                                   <анонимная функция 2>=null])
  ```

- `whenEmpty()` (только класс `Stringable`) — если текущая строка «пуста», вызывает заданную *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова:

  ```
  whenEmpty(<анонимная функция 1>[, <анонимная функция 2>=null])
  ```

  Пример:

  ```
  >>> echo Str::of('')
  ...         ->whenEmpty(function ($s) { return Str::random(10); });
  6vJp3G1h8u
  ```

- `whenNotEmpty()` (только класс `Stringable`) — аналогичен `whenEmpty()`, только вызывает *анонимную функцию 1*, наоборот, если текущая строка не «пуста», а *анонимную функцию 2* — если она «пуста»;
- `whenIsAscii()` (только класс `Stringable`) — если текущая строка содержит только символы из кодировки ASCII, вызывает заданную *анонимную функцию 1*, в противном

случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова такой же, как и у метода `whenEmpty()`;

- `whenIsUuid()` (только класс `Stringable`) — если текущая строка представляет собой универсальный уникальный идентификатор (UUID), вызывает заданную *анонимную функцию 1*, в противном случае — *анонимную функцию 2*. Требования к *анонимным функциям* аналогичны предъявляемым методом `when()`. Формат вызова такой же, как и у метода `whenEmpty()`;
- `tap(<анонимная функция>)` (только класс `Stringable`) — передает текущую строку заданной *анонимной функции* в качестве единственного параметра. Возвращает текущую строку. Обычно используется в цепочках вызовов методов для вывода результатов промежуточных вычислений в целях отладки. Пример:

```
$result = str('Laravel')->prepend('Фреймворк')
        ->tap(function ($str) { dump($str); })->upper();
```

14.2. Обработка массивов

Для обработки массивов применяется набор описанных далее статических методов класса `Illuminate\Support\Arr`.

14.2.1. Добавление, правка и удаление элементов массивов

- `add(<массив>, <ключ>, <значение>)` — добавляет в заданный *массив* элемент с указанными *ключом* и *значением*, если таковой элемент еще не существует в *массиве* или хранит значение `null`. Возвращает массив с добавленным элементом. Примеры:

```
>>> use Illuminate\Support\Arr;
>>> $arr = [0 => 10, 1 => 20, 2 => null];
>>> $arr = Arr::add($arr, 3, 30);
=> [ 0 => 10, 1 => 20, 2 => null, 3 => 30 ]
>>> $arr = Arr::add($arr, 2, 40);
=> [ 0 => 10, 1 => 20, 2 => 40, 3 => 30 ]
>>> $arr = Arr::add($arr, 1, 40);
=> [ 0 => 10, 1 => 20, 3 => 40, 2 => 30 ]
```

- `prepend(<массив>, <значение>[, <ключ>=null])` — добавляет в начало *массива* элемент с заданным *значением* и при необходимости с заданным *ключом*:

```
>>> $arr = Arr::prepend([2, 3, 4], 1);
=> [ 1, 2, 3, 4 ]
>>> $arr = Arr::prepend(['b' => 2, 'c' => 3, 'd' => 4], 1, 'a');
=> [ "a" => 1, "b" => 2, "c" => 3, "d" => 4 ]
```

- `set(<массив>, <ключ>, <значение>)` — заносит заданное *значение* в элемент *массива* с указанным *ключом*. В качестве результата возвращает заданный массив:

```
>>> $arr = ['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4];
>>> Arr::set($arr, 'b', 10000);
=> [ "a" => 1, "b" => 10000, "c" => 3, "d" => 4 ]
```

Можно исправлять значения элементов вложенных массивов, записав пути к их ключам посредством точечной нотации:

```
>>> $arr = ['backend' =>
...             ['platform' =>
...                 ['base' => 'PHP', 'framework' => 'Yii'],
...                 'database' => 'MySQL'],
...          'frontend' => 'Node'];
>>> Arr::set($arr, 'backend.platform.framework', 'Laravel');
=> [ "backend" => [
          "platform" => ["base" => "PHP", 'framework' => 'Laravel'],
          "database" => "MySQL"],
      "frontend" => "Node" ]
```

- `forget(<массив>, <ключ>|<массив ключей>)` — удаляет из заданного *массива* элемент с указанным *ключом* или элементы с ключами, содержащимися в указанном *массиве ключей*. Результата не возвращает. Пример:

```
>>> $arr = ['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4];
>>> Arr::forget($arr, 'c'); $arr;
=> [ "a" => 1, "b" => 2, "d" => 4 ]
>>> Arr::forget($arr, ['a', 'd']); $arr;
=> [ "b" => 2 ]
```

Можно удалять элементы вложенных массивов, записав пути к их ключам посредством точечной нотации:

```
>>> $arr = ['backend' =>
...             ['platform' =>
...                 ['base' => 'PHP', 'framework' => 'Laravel'],
...                 'database' => 'MySQL'],
...          'frontend' => 'Node'];
>>> Arr::forget($arr, 'backend.platform.framework'); $arr;
=> [ "backend" => [
          "platform" => ["base" => "PHP"],
          "database" => "MySQL"],
      "frontend" => "Node" ]
```

- `except(<массив>, <ключ>|<массив ключей>)` — то же самое, что и `forget()`, только возвращает результирующий массив.

Функция-хелпер `data_fill()` по назначению аналогична описанному ранее методу `add()` и имеет такой же формат вызова. Однако она поддерживает точечную нотацию и литерал `*`, обозначающий любой ключ. Примеры:

```
>>> $arr = ['platform' => ['name' => 'PHP', 'version' => 8]];
>>> $arr2 = data_fill($arr, 'platform.version', 7);
=> [ "platform" => ["name" => "PHP", "version" => 8] ]
>>> $arr2 = data_fill($arr, 'platform.edition', 'Windows');
=> [ "platform" => ["name" => "PHP", "version" => 8,
     "edition" => "Windows"] ]

>>> $arr = ['platforms' => [['name' => 'PHP', 'server' => true],
...                         ['name' => 'JavaScript']]];
```

```
>>> $arr2 = data_fill($arr, 'platforms.*.server', false);
=> [ "platforms" => [["name" => "PHP", "server" => true],
                     ["name" => "JavaScript", "server" => false]] ]
```

Функция-хелпер `data_set()` аналогична описанному ранее методу `set()`, но предлагает дополнительные возможности:

```
data_set(<массив>, <ключ>, <значение>[, <перезаписать?>=true])
```

Она поддерживает литерал `*`:

```
>>> $arr = ['platforms' => [['name' => 'PHP', 'server' => true],
...                          ['name' => 'JavaScript', 'server' => false],
...                          ['name' => 'Python']]];
>>> $arr2 = data_set($arr, 'platforms.*.server', true);
=> [ "platforms" => [["name" => "PHP", "server" => true],
                     ["name" => "JavaScript", "server" => true],
                     ["name" => "Python", "server" => true]] ]
```

По умолчанию значения уже существующих в *массиве* элементов будут перезаписаны. Однако, дав параметру *перезаписать* значение `false`, можно отменить их перезапись. Пример:

```
>>> $arr = ['name' => 'PHP', 'server' => false];
>>> data_set($arr, 'version', '8.1');
=> [ "name" => "PHP", "server" => false, "version" => "8.1" ]
>>> data_set($arr, 'server', true);
=> [ "name" => "PHP", "server" => true, "version" => "8.1" ]
>>> data_set($arr, 'name', 'JavaScript', false);
=> [ "name" => "PHP", "server" => true, "version" => "8.1" ]
```

14.2.2. Извлечение элементов массива

- `get()` — возвращает элемент заданного *массива*, имеющий указанный *ключ*. Если элемента с таким *ключом* нет, возвращается *значение по умолчанию*. Формат вызова:

  ```
  get(<массив>, <ключ>[, <значение по умолчанию>=null])
  ```

  Пример:

  ```
  >>> echo Arr::get(['platform' => 'PHP', 'database' => 'MySQL'],
  ...               'database');
  MySQL
  >>> echo Arr::get(['platform' => 'PHP', 'database' => 'MySQL'],
  ...               'frontend', '--- none ---');
  --- none ---
  ```

  Можно извлекать значения элементов вложенных массивов, записав пути к их ключам посредством точечной нотации:

  ```
  >>> echo Arr::get(['platform' =>
  ...                    ['base' => 'PHP', 'framework' => 'Laravel'],
  ...                'database' => 'MySQL'],
  ...      'platform.base');
  PHP
  ```

- `pull()` — возвращает элемент *массива* с заданным *ключом* и удаляет его. Если элемент с таким ключом отсутствует, возвращается *значение по умолчанию*. Формат вызова:

  `pull(<массив>, <ключ>[, <значение по умолчанию>=null])`

  Примеры:

```
>>> $arr = ['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4];
>>> echo Arr::pull($arr, 'b');
2
>>> $arr;
=> [ "a" => 1, "c" => 3, "d" => 4 ]
```

- `where(<массив>, <анонимная функция>)` — возвращает новый массив, составленный из элементов заданного *массива*, для которых указанная *анонимная функция* вернет значение `true`. *Анонимная функция* должна принимать в качестве параметра текущий элемент *массива*. Пример:

```
>>> // Получаем все четные элементы
>>> $arr = Arr::where([2, 81, 934, 679, 45],
...                   function ($el) { return $el % 2 == 0; });
=> [ 0 => 2, 2 => 934 ]
```

- `whereNotNull(<массив>)` — возвращает новый массив, составленный из элементов указанного *массива*, не равных `null`;

- `first()` — возвращает первый элемент *массива*, для которого заданная *анонимная функция* вернет `true`. Сама *функция* должна принимать два параметра: значение очередного элемента *массива* и его индекс (или ключ). Если ни один элемент массива не пройдет проверку, будет возвращено *значение по умолчанию*. Формат вызова:

  `first(<массив>, <анонимная функция>[, <значение по умолчанию>=null])`

  Пример:

```
>>> Извлекаем первый элемент с четным значением
>>> echo Arr::first([1, 2, 3, 4],
...      function ($value, $index) { return $value % 2 == 0; });
2
```

- `last()` — возвращает последний элемент *массива*, для которого заданная *анонимная функция* вернет `true`. Сама *функция* должна принимать два параметра: значение очередного элемента *массива* и его индекс (или ключ). Если ни один элемент массива не пройдет проверку, будет возвращено *значение по умолчанию*. Формат вызова такой же, как и у метода `first()`. Пример:

```
>>> Извлекаем последний элемент с четным значением
>>> echo Arr::last([1, 2, 3, 4],
...      function ($value, $index) { return $value % 2 == 0; });
4
```

- `only(<массив>, <массив ключей>)` — возвращает ассоциативный массив, составленный из элементов заданного *массива*, которые имеют ключи, содержащиеся в *массиве ключей*:

```
>>> $arr = Arr::only(
... ['platform' => 'PHP', 'database' => 'MySQL', 'frontend' => 'Vue'],
... ['platform', 'frontend']);
=> [ "platform" => "PHP", "frontend" => "Vue" ]
```

- `pluck()` — возвращает массив, составленный из всех значений элементов заданного *массива* с указанным *ключом*. Если не указан *ключ элемента, чьи значения будут использованы в качестве ключей результирующего массива*, последний будет индексированным. Формат вызова:

```
pluck(<массив>, <ключ>[, <ключ элемента, чьи значения будут ↵
использованы в качестве ключей элементов результирующего ↵
массива>=null])
```

  Примеры:

```
>>> $arr = [['project' =>
...          ['platform' => 'PHP', 'framework' => 'Laravel']],
...         ['project' =>
...          ['platform' => 'Node', 'framework' => 'Vue']]];
>>> $arr2 = Arr::pluck($arr, 'project.framework');
=> [ "Laravel", "Vue" ]
>>> $arr2 = Arr::pluck($arr, 'project.framework', 'project.platform');
=> [ "PHP" => "Laravel", "Node" => "Vue" ]
```

Функция-хелпер `data_get()` по назначению схожа с описанным ранее методом `get()` и имеет тот же формат вызова, однако поддерживает литерал `*`, обозначающий любой ключ, возвращая в случае его использования массив значений:

```
>>> $arr = ['platforms' => [['name' => 'PHP', 'server' => true],
...                         ['name' => 'JavaScript']]];
>>> $arr2 = data_get($arr, 'platforms.*.name');
=> [ "PHP", "JavaScript" ]
```

Еще две полезные функции-хелперы:

- `head(<массив>)` — возвращает первый элемент заданного *массива*;
- `last(<массив>)` — возвращает последний элемент заданного *массива*.

14.2.3. Проверка существования элементов массивов

- `exists(<массив>, <ключ>)` — возвращает `true`, если в заданном *массиве* присутствует элемент с указанным *ключом*, и `false` — в противном случае:

```
>>> $arr = ['server' => 'Apache', 'platform' => 'PHP'];
>>> echo Arr::exists($arr, 'platform');
true
>>> echo Arr::exists($arr, 'client');
false
```

- `has(<массив>, <ключ>|<массив ключей>)` — возвращает `true`, если в *массиве* присутствует элемент с заданным *ключом* или *все* элементы с ключами, содержащимися в *массиве ключей*, и `false` — в противном случае. Примеры:

```
>>> $arr = ['platform' => 'PHP', 'database' => 'MySQL'];
>>> echo Arr::has($arr, 'database');
true
>>> echo Arr::has($arr, 'frontend');
false
>>> echo Arr::has($arr, ['platform', 'database']);
true
>>> echo Arr::has($arr, ['platform', 'frontend']);
false
```

Можно проверять существование элементов вложенных массивов, записав пути к их ключам посредством точечной нотации:

```
>>> $arr = ['backend' =>
...             ['platform' =>
...                 ['base' => 'PHP', 'framework' => 'Laravel'],
...                 'database' => 'MySQL'],
...          'frontend' => 'Node'];
>>> echo Arr::has($arr, 'backend.platform.base');
true
```

- `hasAny(<`*массив*`>, <`*ключ*`>|<`*массив ключей*`>)` — возвращает `true`, если в *массиве* присутствует элемент с заданным *ключом* или *хотя бы один* из элементов с ключом, содержащимся в *массиве ключей*, и `false` — в противном случае. Пример:

```
>>> echo Arr::hasAny($arr, ['platform', 'frontend']);
true
```

14.2.4. Получение сведений о массиве

- `accessible(<`*значение*`>)` — возвращает `true`, если заданное *значение* является массивом или поддерживает функциональность массива (например, является коллекцией Laravel), и `false` — в противном случае:

```
>>> echo Arr::accessible([1, 2, 3]);
true
>>> echo Arr::accessible(1);
false
```

- `isList(<`*массив*`>)` — возвращает `true`, если заданный *массив* является индексированным, и `false` — в противном случае:

```
>>> echo Arr::isList([1, 2, 3, 4]);
true
>>> echo Arr::isList(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
false
```

- `isAssoc(<`*массив*`>)` — возвращает `true`, если *массив* является ассоциативным, и `false` — в противном случае:

```
>>> echo Arr::isAssoc(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
true
>>> echo Arr::isAssoc([1, 2, 3, 4]);
false
```

14.2.5. Упорядочивание элементов массивов

- `sort(<массив>[, <анонимная функция>=null])` — возвращает массив, составленный из элементов заданного *массива*, чьи элементы отсортированы по возрастанию их значений:

```
>>> $arr = Arr::sort([5, 80, -4, 9, 12]);
=> [ 2 => -4, 0 => 5, 3 => 9, 4 => 12, 1 => 80 ]
```

  При сортировке многомерных массивов необходимо указать *анонимную функцию*, которая должна возвращать значение, по которому будет выполняться сортировка элементов. В качестве параметра она должна принимать очередной элемент сортируемого массива. Пример:

```
>>> $arr = Arr::sort([['id' => 1, 'name' => 'Python'],
...                   ['id' => 2, 'name' => 'JavaScript'],
...                   ['id' => 3, 'name' => 'Ruby'],
...                   ['id' => 4, 'name' => 'PHP']],
...                   function ($el) { return $el['name']; });
=> [ 1 => ["id" => 2, "name" => "JavaScript" ],
     3 => ["id" => 4, "name" => "PHP"],
     0 => ["id" => 1, "name" => "Python"],
     2 => ["id" => 3, "name" => "Ruby"] ]
```

- `sortRecursive(<массив>)` — возвращает массив, составленный из элементов заданного *массива*, чьи элементы отсортированы по возрастанию их значений. Также сортирует элементы вложенных массивов. Пример:

```
>>> $arr = Arr::sortRecursive([[22, 8, -5, 6],
...                            ['PHP', 'JavaScript', 'Python', 'C#']]);
=> [ [-5, 6, 8, 22], ["C#", "JavaScript", "PHP", "Python"] ]
```

- `shuffle(<массив>[, <значение переинициализации>=null])` — случайным образом перемешивает элементы заданного *массива*. Можно указать *значение переинициализации* для встроенного в PHP генератора псевдослучайных чисел. В качестве результата возвращает переупорядоченный *массив*.

14.2.6. Прочие инструменты для обработки массивов

- `divide(<массив>)` — возвращает массив с двумя элементами: массивом из ключей элементов заданного *массива* и массивом из значений его элементов:

```
>>> $arr = Arr::divide(['platform' => 'PHP', 'database' => 'MySQL',
...                     'framework' => 'Laravel']);
=> [ ["platform", "database", "framework"],
     ["PHP", "MySQL", "Laravel"] ]
```

- `collapse(<массив>)` — уменьшает мерность заданного массива на единицу и возвращает полученный в результате массив:

```
>>> Превращаем двумерный массив в одномерный
>>> $arr = Arr::collapse([[1,2], [3, 4, 5], [6, 7]]);
=> [ 1, 2, 3, 4, 5, 6, 7 ]
```

```
>>> А трехмерный — в двумерный
>>> $arr = Arr::collapse([[1,2], [3, [4, 5]], [6, 7]]);
=> [ 1, 2, 3, [4, 5], 6, 7 ]
```

- `flatten(<массив>)` — превращает заданный многомерный *массив* в одномерный и возвращает полученный результат:

```
>>> $arr = Arr::flatten([[1,2], [3, [4, 5]], [6, 7]]);
=> [ 1, 2, 3, 4, 5, 6, 7 ]
```

- `dot(<массив>[, <префикс>=''])` — превращает заданный многомерный ассоциативный *массив* в одномерный, в котором ключи элементов составлены из ключей оригинального *массива* через точку, и возвращает этот массив. Можно указать *префикс*, который будет добавлен к ключам результирующего массива. Примеры:

```
>>> $arr = Arr::dot(['backend' =>
... ['platform' => ['base' => 'PHP', 'framework' => 'Laravel'],
... 'database' => 'MySQL'], 'frontend' => 'Node']);
=> [ "backend.platform.base" => "PHP",
     "backend.platform.framework" => "Laravel",
     "backend.database" => "MySQL",
     "frontend" => "Node" ]
```

- `undot(<массив>)` — превращает заданный одномерный ассоциативный массив, в котором ключи элементов состоят из нескольких слов, разделенных точками, в многомерный, который и возвращает. Фактически выполняет действие, обратное методу `dot()`. Пример:

```
>>> $arr = Arr::undot(["backend.platform.base" => "PHP",
...                    "backend.platform.framework" => "Laravel",
...                    "backend.database" => "MySQL",
...                    "frontend" => "Node"]);
=> [ "backend" =>
         ["platform" => ["base" => "PHP", "framework" => "Laravel"],
          "database" => "MySQL"],
     "frontend" => "Node" ]
```

- `crossJoin(<массив 1>, <массив 2>, ... <массив n>)` — возвращает декартово произведение заданных *массивов*:

```
>>> $arr = Arr::crossJoin([1, 2], ['a', 'b']);
=> [ [1, "a"], [1, "b"], [2, "a"], [2, "b"] ]
```

- `keyBy(<массив>, <ключ>)` — превращает заданный индексированный *массив*, содержащий ассоциативные массивы, в ассоциативный, который и возвращает. В качестве ключа каждого элемента будет использовано значение элемента вложенного массива с заданным *ключом*. Пример:

```
>>> $arr = [['name' => 'Laravel', 'version' => '9'],
...         ['name' => 'Django', 'version' => '4.0']];
>>> Arr::keyBy($arr, 'name');
=> [ "Laravel" => [ "name" => "Laravel", "version" => "9" ],
     "Django" => [ "name" => "Django", "version" => "4.0" ] ]
```

- `query(<массив>)` — на основе элементов *массива* формирует и возвращает строку GET-параметров:

```
>>> echo Arr::query(['platform' => 'PHP', 'framework' => 'Laravel']);
platform=PHP&framework=Laravel
>>> echo Arr::query(['backend' =>
...         ['platform' => 'PHP', 'framework' => 'Laravel'],
...     'frontend' => 'Node']);
backend%5Bplatform%5D=PHP&backend%5Bframework%5D=Laravel&frontend=Node
```

- `random(<массив>[, <количество элементов>=null])` — возвращает массив, содержащий указанное *количество* случайным образом выбранных элементов заданного *массива*. Если *количество* не задано, возвращает один случайно выбранный элемент. Примеры:

```
>>> echo Arr::random([1, 2, 3, 4, 5, 6, 7, 8]);
7
>>> $arr = Arr::random([1, 2, 3, 4, 5, 6, 7, 8], 3);
=> [ 1, 5, 8 ]
```

- `wrap(<значение>)` — если заданное *значение* является:
  - чем-либо отличным от массива и `null`, — превращает его в массив из одного элемента и возвращает в качестве результата:

```
>>> $arr = Arr::wrap('Laravel');
=> [ "Laravel" ]
```

  - массивом — возвращает его без изменений:

```
>>> $arr = Arr::wrap(['Laravel']);
=> [ "Laravel" ]
```

  - `null` — возвращает «пустой» массив;

- `toCssClasses(<массив стилевых классов>)` — возвращает строку, которая составлена из стилевых классов, перечисленных в заданном ассоциативном *массиве*. Очередной стилевой класс включается в результирующую строку лишь в том случае, если указанное у него условие истинно. Ключ каждого элемента *массива* должен соответствовать одному из стилевых классов, а значение элемента задаст необходимое условие. *Массив* может содержать и обычные, индексированные элементы — они будут включены в результирующую строку в любом случае. Пример:

```
>>> $isActive = true;
>>> $hasError = false;
>>> $classes = ['form-text', 'fw-bold' => $isActive,
...             'invalid-feedback' => $hasError];
>>> echo Arr::toCssClasses($classes);
form-text fw-bold
```

14.3. Функции-хелперы

Хелперы — это функции, доступные в любом месте кода сайта и служащие для упрощения программирования. К хелперам, в частности, относятся описанные в предыдущих главах функции: `view()`, `request()`, `response()`, `config()` и др.

В этом разделе описана часть функций-хелперов, выполняющих всевозможные служебные операции. Об остальных хелперах, предоставляемых Laravel, будет рассказано в следующих главах.

14.3.1. Функции, выдающие пути к ключевым папкам

Приведенные здесь функции-хелперы возвращают:

- при вызове без параметра — полный путь к соответствующей папке проекта;
- при указании в качестве параметра относительного *пути к файлу*, находящемуся в соответствующей папке, — полный путь к этому файлу.

Далее идет список функций:

- `base_path(<путь к файлу>)` — возвращает путь к папке проекта:

```
>>> echo base_path();
C:\Projects\sites\bboard
>>> echo base_path('.env');
C:\Projects\sites\bboard\.env
```

- `app_path(<путь к файлу>)` — возвращает путь к папке app:

```
>>> echo app_path();
C:\Projects\sites\bboard\app
>>> echo app_path('http\\controllers\\BbController.php');
C:\Projects\sites\bboard\app\http\controllers\BbController.php
```

- `config_path(<путь к файлу>)` — возвращает путь к папке config;
- `database_path(<путь к файлу>)` — возвращает путь к папке database:

```
>>> echo database_path('data.sqlite');
C:\Projects\sites\bboard\database\data.sqlite
```

- `lang_path(<путь к файлу>)` — возвращает путь к папке lang;
- `public_path(<путь к файлу>)` — возвращает путь к папке public;
- `resource_path(<путь к файлу>)` — возвращает путь к папке resources;
- `storage_path(<путь к файлу>)` — возвращает путь к папке storage.

14.3.2. Служебные функции

- `now()` — возвращает объект класса `Carbon`, хранящий текущую временну́ю отметку (дату и время);
- `today()` — возвращает объект класса `Carbon`, хранящий текущую дату;
- `blank(<значение>)` — возвращает `true`, если заданное *значение* является «пустым», и `false` — в противном случае. «Пустыми» значениями считаются «пустая» строка, строка, состоящая из одних пробелов, «пустая» коллекция и `null`. Примеры:

```
>>> blank('');
=> true
>>> blank('             ');
=> true
```

```
>>> blank('Laravel');
=> false
>>> // Любые числа, даже 0, — не "пусты"
>>> blank(0);
=> false
>>> // Любые логические величины, даже false, — не "пусты"
>>> blank(true);
=> false
>>> blank(false);
=> false
```

- `filled(<значение>)` — возвращает `true`, если заданное *значение* является, наоборот, не «пустым», и `false` — в противном случае (какие значения являются «пустыми», рассказывалось в описании функции `blank()`);
- `transform()` — если заданное *значение* не «пустое» (о «пустых» значениях говорилось в описании функции `blank()`, вызывает указанную *анонимную функцию*, передавая ей *значение* в качестве параметра, и возвращает выданный *функцией* результат. Если заданное *значение* «пустое», возвращает указанное *значение по умолчанию*. Формат вызова:

  `transform(<значение>, <анонимная функция>[, <значение по умолчанию>=null])`

  Примеры:

```
>>> transform('admin@bboard.ru', function ($email) {
...     return 'Email: ' . $email;
... }, '---');
=> "Email: admin@bboard.ru"

>>> transform('', function ($email) {
...     return 'Email: ' . $email;
... }, '---');
=> "---"
```

  Вместо *значения по умолчанию* можно указать анонимную функцию, которая примет в качестве параметра указанное *значение*. Возвращаемый ею результат будет возвращен функцией `transform()`. Пример:

```
>>> transform('', function ($email) {
...     return 'Email: ' . $email;
... }, function ($value) {
...     return '(' . $value . ')';
... });
=> "()"
```

- `tap(<объект>[, <анонимная функция>=null])` — позволяет вызвать у заданного *объекта* любой метод таким образом, чтобы он в качестве результата в любом случае вернул заданный *объект*:

```
// Метод fill() записи модели сам по себе не возвращает результата,
// но, используя функцию tap(), можно заставить его возвращать
// текущую запись
```

```
tap($bb)->fill(['price' => 20000000])
        ->fill(['content' => 'Большой, хороший']);
```

Если указана *анонимная функция*, она будет вызвана и получит в качестве параметра заданный *объект*. Результат, возвращенный этой *функцией*, игнорируется. Пример:

```
tap($bb, function ($rec) { $rec->fill(['price' => 20000000]); })
         ->fill(['content' => 'Большой, хороший']);
```

- `optional()` — позволяет обратиться к любому свойству или методу указанного *объекта*, даже если вместо него задано значение `null`, без возникновения исключения. Если вместо объекта задано `null`, в результате обращения к любому свойству или методу выдается `null`. Формат вызова:

  `optional(<объект или null>[, <анонимная функция>=null])`

  Пример:

```
>>> use App\Models\User;
>>> // Ищем какого-либо существующего пользователя
>>> $user = User::firstWhere('name', 'admin');
>>> optional($user)->email;
=> "admin@bboard.ru"
>>> // Ищем гарантированно не существующего пользователя.
>>> // В этом случае метод firstWhere() вернет null.
>>> $user = User::firstWhere('name', 'superadmin');
>>> optional($user)->email;
=> null
```

  Можно указать *анонимную функцию*, принимающую один параметр. Тогда, если указан *объект*, *анонимная функция* будет вызвана с передачей ей *объекта* в качестве параметра, и метод `optional()` вернет результат, возвращенный этой *анонимной функцией*. Пример:

```
>>> $user = User::firstWhere('name', 'admin');
>>> optional($user, function ($u) { return $u->email; });
=> "admin@bboard.ru"
```

- `value(<значение>|<анонимная функция>)` — если указано *значение*, оно возвращается без изменений. Если задана не принимающая параметров *анонимная функция*, она будет вызвана, и выданный ею результат будет возвращен;
- `with(<значение>, [<анонимная функция>=null])` — если *анонимная функция* не указана, возвращается заданное *значение*. В противном случае вызывается *анонимная функция* с передачей ей в качестве параметра заданного *значения* и возвращается выданный *функцией* результат;
- `class_basename(<путь к классу>)` — возвращает имя класса, извлеченное из заданного *пути*:

```
>>> echo class_basename('App\\Http\\Controllers\\MainController');
MainController
```

- `class_uses_recursive(<объект>|<путь к классу>)` — возвращает перечень всех трейтов, использованных заданным классом, включая трейты, использованные всеми его

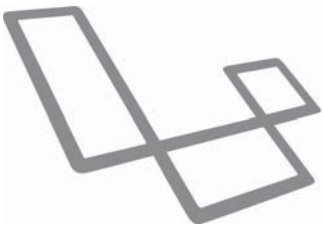
суперклассами и использованными им трейтами. Класс может быть указан в качестве созданного на его основе *объекта* или в виде строки с *путем к нему*. Возвращаемый перечень представляет собой ассоциативный массив, ключи и значения элементов которого являются путями к трейтам. Примеры:

```
>>> class_uses_recursive('App\\Http\\Controllers\\MainController');
=> [ "Illuminate\Foundation\Auth\Access\AuthorizesRequests" =>
     "Illuminate\Foundation\Auth\Access\AuthorizesRequests",
     "Illuminate\Foundation\Bus\DispatchesJobs" =>
     "Illuminate\Foundation\Bus\DispatchesJobs",
     "Illuminate\Foundation\Validation\ValidatesRequests" =>
     "Illuminate\Foundation\Validation\ValidatesRequests" ]

>>> use App\Models\User;
>>> $user = User::first();
>>> class_uses_recursive($user);
=> [ "Illuminate\Database\Eloquent\Concerns\HasAttributes" =>
     "Illuminate\Database\Eloquent\Concerns\HasAttributes",
     "Illuminate\Database\Eloquent\Concerns\HasEvents" =>
     "Illuminate\Database\Eloquent\Concerns\HasEvents",
     . . .
     "Illuminate\Notifications\RoutesNotifications" =>
     "Illuminate\Notifications\RoutesNotifications" ]
```

- `trait_uses_recursive(<путь к трейту>)` — возвращает перечень всех трейтов, использованных трейтом с заданным *путем*, включая трейты, использованные всеми использованными им трейтами. Возвращаемый перечень представляет собой ассоциативный массив, ключи и значения элементов которого являются путями к трейтам. Пример:

```
>>> trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
=> [ "Illuminate\Notifications\HasDatabaseNotifications" =>
     "Illuminate\Notifications\HasDatabaseNotifications",
     "Illuminate\Notifications\RoutesNotifications" =>
     "Illuminate\Notifications\RoutesNotifications" ]
```



ГЛАВА 15

Коллекции Laravel

Коллекция Laravel — это удобная «обертка» вокруг обычного массива PHP. Коллекция может быть индексированной, ассоциативной или комбинированной, одно- или многомерной.

15.1. Обычные коллекции

Обычная коллекция хранит все свои элементы в оперативной памяти и представляется объектом класса `Illuminate\Support\Collection`.

15.1.1. Создание обычных коллекций

Создать коллекцию на основе заданного *массива* можно тремя способами:

- непосредственно вызовом конструктора класса `Collection` в формате `Collection([<`*массив*`>=[]])`:

```
>>> use Illuminate\Support\Collection;
>>> $coll = new Collection([1, 2, 3]);
```

- вызовом статического метода `make(<`*массив*`>)` класса `Collection`:

```
>>> $coll = Collection::make([1, 2, 3]);
```

- вызовом функции-хелпера `collect(<`*массив*`>)`:

```
>>> $coll = collect([1, 2, 3]);
```

Класс поддерживает несколько методов, которые могут пригодиться при создании коллекций:

- `times(<`*количество*`>[, <`*анонимная функция*`>=null])` — статический, вызывает указанную *анонимную функцию* заданное *количество* раз, передавая ей в качестве параметра число от 1 до заданного *количества*, добавляет возвращенные *анонимной функцией* результаты в новую коллекцию и возвращает ее в качестве результата:

```
>>> $coll = Collection::times(10, function ($num) {
...     return $num ** 2;
... });
=> [ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

Если *анонимная функция* не указана, возвращает коллекцию чисел от 1 до заданного *количества*:

```
>>> $coll = Collection::times(10);
=> [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

- `range(<начальное число>, <конечное число>)` — статический, возвращает коллекцию, содержащую набор последовательно увеличивающихся целых чисел от *начального* до *конечного* включительно;
- `wrap(<значение>)` — статический. Если заданное *значение* не является коллекцией, возвращает новую коллекцию с единственным элементом, хранящим это *значение*. Если же *значение* является коллекцией, возвращает ее без изменений;
- `combine(<коллекция значений>)` — создает новую ассоциативную коллекцию, используя значения элементов текущей коллекции как ключи, а значения из заданной *коллекции значений* — как значения элементов, и возвращает ее в качестве результата:

```
>>> $coll = collect(['platform', 'database']);
>>> $coll2 = $coll->combine(['PHP', 'MySQL']);
=> [ "platform" => "PHP", "database" => "MySQL" ]
```

- `collect()` — возвращает копию текущей коллекции.

15.1.2. Добавление, правка и удаление элементов коллекции

- `add(<элемент>)` — добавляет в конец текущей индексированной коллекции новый *элемент*. В качестве результата возвращает текущую коллекцию;
- `push(<элемент 1>, <элемент 2>, ... <элемент n>)` — добавляет в конец текущей индексированной коллекции заданные *элементы*. В качестве результата возвращает текущую коллекцию;
- `prepend(<элемент>[, <ключ>=null])` — добавляет в начало текущей коллекции новый *элемент*. Если текущая коллекция ассоциативная, следует указать *ключ* для нового элемента. В качестве результата возвращает текущую коллекцию;
- `put(<ключ>, <значение>)` — задает новое *значение* элемента с указанным *ключом* текущей ассоциативной коллекции. Если элемент с таким *ключом* отсутствует, он будет создан. В качестве результата возвращает текущую коллекцию;
- `pop()` — удаляет и возвращает последний элемент текущей коллекции;
- `shift()` — удаляет и возвращает первый элемент текущей коллекции;
- `pull(<ключ>[, <значение по умолчанию>=null])` — удаляет и возвращает элемент с указанным *ключом* текущей ассоциативной коллекции. Если элемент с таким *ключом* отсутствует, возвращается *значение по умолчанию*;
- `concat(<добавляемая коллекция>)` — добавляет элементы из указанной *коллекции* (также можно указать массив) в конец текущей коллекции и возвращает последнюю в качестве результата:

```
>>> $coll = collect(['Дом'])->concat(collect(['Дача', 'Грузовик']))
...                         ->concat(['Лопата', 'Мотыга']);
=> [ "Дом", "Дача", "Грузовик", "Лопата", "Мотыга" ]
```

- `merge(<добавляемая коллекция>)` — если текущая и *добавляемая* коллекции индексированные, ведет себя так же, как и метод `concat()`. Если обе коллекции ассоциативные, также заменяет значения элементов текущей коллекции значениями элементов *добавляемой* коллекции, имеющих те же строковые ключи. Результат возвращает в виде новой коллекции. Примеры:

```
>>> $coll = collect(['platform' => 'PHP'])->merge(['server' => true]);
=> [ "platform" => "PHP", "server" => true ]
>>> $coll = collect(['platform' => 'PHP'])
...       ->merge(['platform' => 'Python', 'server' => true]);
=> [ "platform" => "Python", "server" => true ]
```

- `union(<добавляемая коллекция>)` — то же самое, что и `merge()`, только элементы текущей коллекции сохраняют свои значения:

```
>>> $coll = collect(['platform' => 'PHP'])
...       ->union(['platform' => 'Python', 'server' => true]);
=> [ "platform" => "PHP", "server" => true ]
```

- `mergeRecursive(<добавляемая коллекция>)` — то же самое, что и `merge()`, только если в обеих коллекциях есть элементы с одинаковыми строковыми ключами, их значения объединяются в один массив. Пример:

```
>>> $coll = collect(['platform' => 'PHP', 'side' => 'server'])
...  ->mergeRecursive(['platform' => 'JavaScript', 'side'=>'client']);
=> [ "platform" => ["PHP", "JavaScript"],
     "side" => ["server", "client"] ]
```

- `replace(<добавляемая коллекция>)` — то же самое, что и `merge()`, только дополнительно заменяет значения элементов текущей коллекции с совпадающими числовыми ключами;
- `replaceRecursive(<добавляемая коллекция>)` — то же самое, что и `mergeRecursive()`, только дополнительно заменяет значения элементов текущей коллекции с совпадающими числовыми ключами;
- `forget(<ключ>|<массив ключей>)` — удаляет из текущей коллекции элемент с заданным *ключом* или элементы с ключами, содержащимися в *массиве*. Возвращает текущую коллекцию. Примеры:

```
>>> $coll = collect(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
>>> $coll->forget('b');
=> [ "a" => 1, "c" => 3, "d" => 4 ]
>>> $coll->forget(['a', 'c']);
=> [ "d" => 4 ]
```

- `pad(<размер>, <значение>)` — увеличивает размер текущей коллекции до заданной в первом параметре величины, добавляя новые элементы и занося в них указанное *значение*. Если заданный *размер* положительный, новые элементы добавляются в конце коллекции, если отрицательный — в начале. Результат возвращается в виде

новой коллекции. Если заданный *размер* меньше или равен размеру текущей коллекции, ничего не делает. Примеры:

```
>>> $coll = collect([1, 2, 3])->pad(5, 0);
=> [ 1, 2, 3, 0, 0 ]
>>> $coll = collect([1, 2, 3])->pad(-5, 0);
=> [ 0, 0, 1, 2, 3 ]
```

15.1.3. Извлечение отдельных элементов и частей коллекции

- `get(<`*ключ*`>[, <`*значение по умолчанию*`>=null])` — возвращает элемент текущей ассоциативной коллекции с заданным *ключом*. Если такого элемента нет, возвращает *значение по умолчанию*:

  `get(<`*ключ*`>[, <`*значение по умолчанию*`>|<`*анонимная функция*`>=null])`

  Примеры:

  ```
  >>> $coll = collect(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
  >>> echo $coll->get('b');
  2
  >>> echo $coll->get('efgh', 10000);
  10000
  ```

  Вместо *значения по умолчанию* можно указать *анонимную функцию*, не принимающую параметров и возвращающую результат, который и будет выдан методом `get()`:

  ```
  >>> echo $coll->get('efgh', function () { return 10 ** 4; });
  10000
  ```

- `slice(<`*номер первого элемента*`>[, <`*количество элементов*`>=null])` — возвращает в виде новой коллекции фрагмент текущей коллекции, начинающийся с элемента с заданным *номером* и содержащий заданное *количество* элементов. Если *количество* не указано, во фрагмент войдут все оставшиеся элементы текущей коллекции. Примеры:

  ```
  >>> $coll = collect([1, 2, 3, 4, 5]);
  >>> $col2 = $coll->slice(2);
  => [ 2 => 3, 3 => 4, 4 => 5 ]
  >>> $col2 = $coll->slice(2, 2);
  => [ 2 => 3, 3 => 4 ]
  ```

- `splice()` — то же самое, что и `slice()`, только удаляет из текущей коллекции возвращенный фрагмент или заменяет его указанным *массивом*:

  ```
  splice(<номер первого элемента>[, <количество элементов>=null[,
                                    <заменяющий массив>=[]]])
  ```

  Примеры:

  ```
  >>> $coll = collect([1, 2, 3, 4, 5, 6, 7, 8]);
  >>> $col2 = $coll->splice(2, 3);
  => [ 3, 4, 5 ]
  ```

```
>>> $coll;
=> [ 1, 2, 6, 7, 8 ]
>>> $col2 = $coll->splice(3, 2, [10, 11, 12, 13]);
=> [ 7, 8 ]
>>> $coll;
=> [ 1, 2, 6, 10, 11, 12, 13 ]
```

- `keys()` — возвращает новую коллекцию, содержащую ключи элементов текущей ассоциативной коллекции;
- `values()` — возвращает новую коллекцию, содержащую значения элементов текущей ассоциативной коллекции;
- `pluck(<`*ключ 1*`>[, <`*ключ 2*`>=null])` — возвращает индексированную коллекцию, содержащую все значения элементов текущей ассоциативной коллекции с заданным *ключом 1*. Если указан *ключ 2*, будет возвращена ассоциативная коллекция, ключи элементов которой будут взяты из элементов текущей коллекции с *ключом 2*. Примеры:

```
>>> $coll = collect([
...       ['id' => 1, 'platform' => 'PHP', 'side' => 'server'],
...       ['id' => 2, 'platform' => 'JavaScript', 'side' => 'client'],
...       ['id' => 3, 'platform' => 'Python', 'side' => 'server']]);
>>> $coll2 = $coll->pluck('platform');
=> [ "PHP", "JavaScript", "Python" ]
>>> $coll2 = $coll->pluck('platform', 'id');
=> [ 1 => "PHP", 2 => "JavaScript", 3 => "Python" ]
```

- `nth(<`*n*`>[, <`*смещение*`>=0])` — возвращает новую коллекцию, составленную из каждого *n*-го элемента текущей коллекции. Можно указать *смещение* от начала коллекции, выраженное в количестве элементов. Примеры:

```
>>> $coll = collect([1, 2, 3, 4, 5, 6, 7, 8])->nth(3);
=> [ 1, 4, 7 ]
>>> $coll = collect([1, 2, 3, 4, 5, 6, 7, 8])->nth(3, 1);
=> [ 2, 5, 8 ]
```

- `only(<`*массив ключей*`>)` — возвращает коллекцию, составленную из элементов текущей коллекции, чьи ключи приведены в *массиве*:

```
>>> $coll = collect(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
>>> $coll2 = $coll->only(['b', 'c']);
=> [ "b" => 2, "c" => 3 ]
```

- `except(<`*массив ключей*`>)` — возвращает коллекцию, составленную из всех элементов текущей коллекции, за исключением элементов с приведенными в *массиве* ключами:

```
>>> $coll2 = $coll->except(['b', 'c']);
=> [ "a" => 1, "d" => 4 ]
```

- `skip(<`*количество*`>)` — убирает из текущей коллекции указанное *количество* начальных элементов и возвращает новую коллекцию, содержащую оставшиеся элементы:

```
>>> $coll = collect([1, 2, 3, 4, 5])->skip(3);
=> [ 3 => 4, 4 => 5 ]
```

- `skipWhile(<`*анонимная функция*`>)` — убирает из текущей коллекции элемент за элементом, пока заданная *анонимная функция*, получив значение очередного элемента, не вернет `false`. Возвращает новую коллекцию, содержащую оставшиеся элементы. Пример:

```
>>> $coll = collect([1, 2, 3, 4, 5])->skipWhile(function ($value) {
...     return sqrt($value) < 2;
... });
=> [ 3 => 4, 4 => 5 ]
```

- `skipUntil(<`*значение*`>|<`*анонимная функция*`>)` — убирает из текущей коллекции элемент за элементом, пока не встретится элемент с заданным *значением* или пока заданная *анонимная функция*, получив значение очередного элемента, не вернет `true`. Возвращает новую коллекцию, содержащую оставшиеся элементы. Примеры:

```
>>> $coll = collect([1, 2, 3, 4, 5])->skipUntil(4);
=> [ 3 => 4, 4 => 5 ]
>>> $coll = collect([1, 2, 3, 4, 5])->skipUntil(function ($value) {
...     return sqrt($value) > 2;
... });
=> [ 4 => 5 ]
```

- `take(<`*количество*`>)` — возвращает новую коллекцию, содержащую указанное *количество* начальных (если *количество* положительное) или конечных (если *количество* отрицательное) элементов текущей коллекции:

```
>>> $coll = collect([1, 2, 3, 4, 5, 6])->take(3);
=> [ 1, 2, 3 ]
>>> $coll = collect([1, 2, 3, 4, 5, 6])->take(-4);
=> [ 2 => 3, 3 => 4, 4 => 5, 5 => 6 ]
```

- `takeWhile(<`*анонимная функция*`>)` — помещает в новую коллекцию один элемент текущей коллекции за другим, пока заданная *анонимная функция*, получив значение очередного элемента, не вернет `false`. Возвращает созданную коллекцию в качестве результата. Пример:

```
>>> $coll = collect([1, 2, 3, 4, 5, 6]);
>>> $coll2 = $coll->takeWhile(function ($value) {
...     return sqrt($value) < 2;
... });
=> [ 1, 2, 3 ]
```

- `takeUntil(<`*значение*`>|<`*анонимная функция*`>)` — помещает в новую коллекцию один элемент текущей коллекции за другим, пока не встретится элемент с заданным *значением* или пока заданная *анонимная функция*, получив значение очередного элемента, не вернет `true`. Возвращает созданную коллекцию в качестве результата. Пример:

```
>>> $coll2 = $coll->takeUntil(4);
=> [ 1, 2, 3 ]
>>> $coll2 = $coll->takeUntil(function ($value) {
...     return sqrt($value) > 2;
... });
=> [ 1, 2, 3, 4 ]
```

- `random([<размер>=null])` — если *размер* не указан, возвращает случайно выбранный элемент текущей коллекции. В противном случае возвращает коллекцию заданного *размера*, содержащую случайно выбранные элементы текущей коллекции. Примеры:

```
>>> $coll = collect(['PHP', 'JavaScript' ,'Python', 'Ruby', 'C#']);
>>> echo $coll->random();
Python
>>> $coll2 = $coll->random(3);
=> [ "JavaScript", "Ruby", "C#" ]
```

  Если заданный *размер* меньше размера текущей коллекции, возбуждается исключение `InvalidArgumentException`;

- `intersect(<коллекция>)` — возвращает коллекцию с элементами текущей коллекции, присутствующими в заданной *коллекции* (также можно указать обычный массив). Работает с индексированными коллекциями. Пример:

```
>>> collect([1, 2, 3, 4])->intersect([2, 4, 6, 8]);
=> [ 1 => 2, 3 => 4 ]
```

- `intersectByKeys(<коллекция>)` — то же самое, что и `intersect()`, только работает с ассоциативными коллекциями и сравнивает лишь ключи элементов:

```
>>> collect(['a' => 1, 'c' => 2])
... ->intersectByKeys(['b' => 2, 'c' => 4]);
=> [ "c" => 2 ]
```

- `diff(<коллекция>)` — возвращает коллекцию с элементами текущей коллекции, отсутствующими в заданной *коллекции* (также можно указать обычный массив). Работает с индексированными коллекциями. Пример:

```
>>> collect([1, 2, 3, 4])->diff([2, 4, 6, 8]);
=> [ 0 => 1, 2 => 3 ]
```

- `diffKeys(<коллекция>)` — то же самое, что и `diff()`, только работает с ассоциативными коллекциями и сравнивает лишь ключи элементов:

```
>>> collect(['a' => 1, 'c' => 2])->diffKeys(['b' => 2, 'c' => 4]);
=> [ "a" => 1 ]
```

- `diffAssoc(<коллекция>)` — то же самое, что и `diff()`, только работает с ассоциативными коллекциями и сравнивает как ключи, так и значения элементов:

```
>>> collect(['a' => 1, 'c' => 2])->diffAssoc(['b' => 2, 'c' => 4]);
=> [ "a" => 1, "c" => 2 ]
```

- `unique()` — возвращает массив, содержащий уникальные элементы текущей коллекции:

```
unique([<ключ>|<анонимная функция>=null[, <строгое сравнение>=false]])
```

  Пример:

```
>>> $coll2 = collect([1, 3, 1, 3, 70, 1, 70])->unique();
=> [ 0 => 1, 1 => 3, 4 => 70 ]
```

  Если текущая коллекция содержит ассоциативные массивы, следует указать *ключ* элемента этих массивов, используемого для проверки на дублирование элементов:

```
>>> $coll = collect([['title' => 'Дом', 'price' => 10000000],
...                  ['title' => 'Конура', 'price' => 200],
...                  ['title' => 'Дом', 'price'=> 50000000]]);
>>> $coll2 = $coll->unique('title');
=> [ ["title" => "Дом", "price" => 10000000],
     ["title" => "Конура", "price" => 200] ]
```

Также можно указать *анонимную функцию*, принимающую значение очередного элемента текущей коллекции и возвращающую значение, используемое для проверки на уникальность элементов:

```
>>> $coll2 = $coll->unique(function ($el) { return $el['title']; });
=> [ ["title" => "Дом", "price" => 10000000],
     ["title" => "Конура", "price" => 200] ]
```

Метод `unique()` выполняет обычное (нестрогое, без учета типа данных) сравнение элементов:

```
>>> $coll2 = collect(['1', 3, 1, '3', 70, 1, '70'])->unique();
=> [ 0 => "1", 1 => 3, 4 => 70 ]
```

Чтобы он выполнял строгое сравнение, необходимо дать параметру *строгое сравнение* значение `true`:

```
>>> $coll2 = collect(['1', 3, 1, '3', 70, 1, '70'])
...        ->unique(null, true);
=> [ 0 => "1", 1 => 3, 2 => 1, 3 => "3", 4 => 70, 6 => "70" ]
```

- `uniqueStrict([<ключ>|<анонимная функция>=null])` — аналогичен `unique()`, только выполняет строгое (с учетом типа данных) сравнение элементов:

```
>>> $coll2 = collect(['1', 3, 1, '3', 70, 1, '70'])->uniqueStrict();
=> [ 0 => "1", 1 => 3, 2 => 1, 3 => "3", 4 => 70, 6 => "70" ]
```

- `duplicates()` — возвращает массив, содержащий дублирующиеся элементы текущей коллекции. Формат вызова такой же, как и у метода `unique()`. Примеры:

```
>>> $coll2 = collect([1, 3, 1, 3, 70])->duplicates();
=> [ 2 => 1, 3 => 3 ]
>>> $coll2 = $coll->duplicates('title');
=> [ 2 => "Дом" ]
>>> $coll2 = $coll->duplicates(function ($el) {
...     return $el['title'];
... });
=> [ 2 => "Дом" ]
```

Метод `duplicates()` выполняет обычное (нестрогое) сравнение элементов:

```
>>> $coll2 = collect([1, 3, 1, '3', 70])->duplicates();
=> [ 2 => 1, 3 => "3" ]
```

Чтобы он выполнял строгое сравнение, необходимо дать параметру *строгое сравнение* значение `true`:

```
>>> $coll2 = collect([1, 3, 1, '3', 70])->duplicates(null, true);
=> [ 2 => 1 ]
```

- `duplicatesStrict()` — аналогичен `duplicate()`, только выполняет строгое сравнение элементов. Формат вызова такой же, как и у метода `uniqueStrict()`. Пример:

```
>>> $coll = collect([1, 3, 1, '3', 70])->duplicatesStrict();
=> [ 2 => 1 ]
```

- `chunk(<размер>)` — разбивает текущую коллекцию на части (*чанки*) указанного *размера*, также представленные коллекциями, помещает их в новую коллекцию и возвращает ее в качестве результата:

```
>>> $coll = collect([1, 2, 3, 4, 5])->chunk(3);
=> [ [1, 2, 3], [3 => 4, 4 => 5] ]
```

  Этот метод может пригодиться при выводе коллекций на странице с применением сетки Bootstrap;

- `chunkWhile(<анонимная функция>)` — разбивает текущую коллекцию на *чанки*, также представленные коллекциями, помещает их в новую коллекцию и возвращает ее в качестве результата. Указанная *анонимная функция* должна принимать три параметра: значение очередного элемента текущей коллекции, его ключ и текущий чанк. Если *функция* вернет `true`, очередной элемент будет помещен в текущий чанк, если `false` — в новый чанк. Пример разбиения коллекций на чанки, в которых максимальная разность между значениями элементов не превышает 5 (метод `first()` коллекции возвращает первый ее элемент):

```
$col1 = collect([1, 6, 9, 11, 17, 23, 29, 32]);
>>> $col2 = $col1->chunkWhile(function ($value, $key, $prevChunk) {
...     return $value < $prevChunk->first() + 5;
... });
=> [ [1, 6], [1 => 9, 2 => 11], [3 => 17], [4 => 23],
     [5 => 29, 7 => 32] ]
```

- `forPage(<номер части>, <количество элементов>)` — извлекает из текущей коллекции часть с заданными *номером* (нумерация начинается с единицы) и максимальным *количеством элементов* и возвращает в виде новой коллекции:

```
>>> $coll = collect([1, 2, 3, 4, 5]);
>>> $part1 = $coll->forPage(1, 3);
=> [ 1, 2, 3 ]
>>> $part1 = $coll->forPage(2, 3);
=> [ 3 => 4, 4 => 5 ]
```

- `split(<количество чанков>)` — разбивает текущую коллекцию на указанное *количество* чанков одинакового по возможности размера, и возвращает результат в виде двумерной коллекции:

```
>>> $coll = collect([1, 2, 3, 4, 5, 6, 7])->split(3);
=> [ [1, 2, 3], [4, 5], [6, 7] ]
```

- `splitIn(<количество чанков>)` — аналогичен `split()`, только делает все чанки, кроме последнего, максимально заполненными:

```
>>> $coll = collect([1, 2, 3, 4, 5, 6, 7])->splitIn(3);
=> [ [1, 2, 3], [4, 5, 6], [7] ]
```

15.1.4. Получение сведений об элементах коллекции

- has(<*ключ*>|<*массив ключей*>) — возвращает true, если в текущей коллекции присутствует элемент с заданным *ключом* или все элементы с ключами, содержащимися в *массиве*, и false — в противном случае:

```
$coll = collect(['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4]);
>>> echo $coll->has('c');
true
>>> echo $coll->has(['c', 'd']);
true
>>> echo $coll->has(['c', 'd', 'f']);
false
```

- contains() — возвращает true, если хотя бы один элемент, удовлетворяющий заданным условиям, присутствует в текущей коллекции, и false — в противном случае. Поддерживаются три формата вызова:

```
contains(<значение>)
contains(<ключ>[, <оператор сравнения>], <значение>)
contains(<анонимная функция>)
```

Первый формат позволяет выяснить, присутствует ли в текущей коллекции элемент, хранящий заданное *значение*, и применяется в случае одномерных коллекций, как индексированных, так и ассоциативных:

```
>>> $coll1 = collect([1, 2, 3, 4, 5]);
>>> $coll1->contains(2);
=> true
>>> $coll1->contains(20);
=> false
```

При вызове в первом формате метод contains() использует обычное (нестрогое) сравнение элементов:

```
>>> $coll1->contains('2');
=> true
```

Второй формат применяется, если текущая коллекция содержит вложенные ассоциативные массивы, и позволяет выяснить, существует ли элемент с указанным *ключом* и значением, сравнение которого с заданным *значением* с использованием указанного *оператора сравнения* окажется успешным. Можно указать любой *оператор сравнения*, поддерживаемый PHP, если же он не указан, будет использован оператор ==. Примеры:

```
>>> $coll2 = collect([['title' => 'Дом', 'price' => 10000000],
...                   ['title' => 'Конура', 'price' => 200]]);
>>> $coll2->contains('price', 200);
=> true
>>> $coll2->contains('price', '==', 200);
=> true
>>> $coll2->contains('price', '<', 200);
=> false
```

Третий формат позволяет реализовать логику поиска элементов в заданной *анонимной функции*. Она должна принимать в качестве параметров значение и индекс (или ключ) очередного элемента текущей коллекции и возвращать `true`, если этот элемент удовлетворяет заданным условиям поиска, и `false` — в противном случае. Пример:

```
>>> // Ищем дом по цене менее 5 млн
>>> $coll2->contains(function ($value, $key) {
...     return $value['title'] == 'Дом' && $value['price'] < 5000000;
... } );
=> false
```

- `some()` — то же самое, что и `contains()`;
- `containsStrict()` — то же самое, что и `contains()`, только при вызове в первом формате выполняет строгое сравнение элементов. Поддерживаются три формата вызова:

```
containsStrict(<значение>)
containsStrict(<ключ>, <значение>)
containsStrict(<анонимная функция>)
```

Пример:

```
>>> $coll1->containsStrict(2);
=> true
>>> $coll1->containsStrict('2');
=> false
```

- `doesntContain()` — аналогичен `contains()`, только возвращает `true`, если, наоборот, ни один элемент, удовлетворяющий заданным условиям, не присутствует в текущей коллекции, и `false` — в противном случае;
- `every()` — возвращает `true`, если *все* элементы текущей коллекции удовлетворяют заданным условиям, и `false` — в противном случае. Форматы вызова такие же, как и у метода `contains()`. Примеры:

```
>>> $coll = collect([['title' => 'Дом', 'price' => 10000000],
...                  ['title' => 'Конура', 'price' => 200],
...                  ['title' => 'Дом', 'price' => 50000000]]);
>>> echo $coll->every('price', '>', 100);
true
>>> echo $coll->every(function ($value, $index) {
...     return $value['price'] < 100;
... });
false
```

15.1.5. Перебор элементов коллекции

- `each(<анонимная функция>)` — перебирает элементы текущей коллекции и для каждого вызывает заданную *анонимную функцию*. Последняя должна принимать в качестве параметров значение очередного элемента и его индекс (ключ). Возвращает текущую коллекцию. Пример:

```
>>> collect([1, 2, 3, 4])->each(function ($value, $index) {
...     echo $index, ': ', $value, ' | ';
... });
0: 1 | 1: 2 | 2: 3 | 3: 4 |
```

Чтобы прервать перебор коллекции, следует вернуть из *анонимной функции* значение `false`;

- `eachSpread(<`*анонимная функция*`>)` — аналогичен `each()`, только применяется, если текущая коллекция хранит вложенные массивы, и передает в *анонимную функцию* значения элементов очередного вложенного массива и его индекс (ключ):

```
>>> collect([['Дом', 10000000], ['Конура', 200]])
... ->eachSpread(function ($title, $price, $index) {
...     echo $index, ': ', $title, ' (', $price, ') | ';
... });
0: Дом (10000000) | 1: Конура (200) |
```

15.1.6. Поиск и фильтрация элементов коллекции

- `search()` — ищет элемент текущей коллекции, удовлетворяющий заданным условиям, и возвращает его индекс (ключ). Если подходящий элемент не найден, возвращается `false`. Поддерживает два формата вызова:

```
search(<искомое значение>[, <строгое сравнение?>=false])
search(<анонимная функция>)
```

При вызове в первом формате метод ищет первый элемент, имеющий заданное *значение*. По умолчанию выполняется нестрогое сравнение значений, а чтобы задействовать строгое, следует дать параметру *строгое сравнение* значение `true`. Примеры:

```
>>> $coll = collect([1, 67, -5, 274]);
>>> echo $coll->search(-5);
2
>>> echo $coll->search(100);
false
>>> echo $coll->search(1);
0
>>> echo $coll->search('1');
0
>>> echo $coll->search('1', true);
false
```

При вызове во втором формате метод ищет первый элемент, для которого заданная *анонимная функция* вернет `true`. *Анонимная функция* должна принимать значение и индекс (ключ) очередного элемента коллекции. Пример:

```
>>> echo $coll->search(function ($value, $index) {
...     return $value === 1;
... });
0
```

```
>>> echo $coll->search(function ($value, $index) {
...       return $value === '1';
... });
false
```

- `first()` — возвращает первый элемент текущей коллекции, для которого заданная *анонимная функция* вернет `true`. *Анонимная функция* должна принимать в качестве параметров значение и индекс (ключ) очередного элемента текущей коллекции. Если подходящих элементов в коллекции нет, будет возвращено *значение по умолчанию*. Формат вызова:

  ```
  first([<анонимная функция>=null[, <значение по умолчанию>=null]])
  ```

  Пример:

  ```
  >>> collect([['title' => 'Дом', 'price' => 10000000],
  ...          ['title' => 'Конура', 'price' => 200]])
  ... ->first(function ($value, $index) {
  ...     return $value['price'] > 2000000;
  ... });
  => [ "title" => "Дом", "price" => 10000000 ]
  ```

  Если *анонимная функция* не указана, будет возвращен первый элемент текущей коллекции;

- `last()` — аналогичен методу `first()`, только возвращает последний найденный элемент текущей коллекции;

- `firstWhere(<ключ>[, <оператор сравнения>][, <значение>])` — возвращает первый элемент текущей двумерной коллекции, в котором сравнение элемента с указанным *ключом* и заданного *значения* с применением указанного *оператора сравнения* оказалось успешным. Можно указать любой *оператор сравнения*, поддерживаемый PHP, если же он не указан, будет использован оператор `==`. Если подходящий элемент не найден, возвращается `null`. Примеры:

  ```
  >>> $coll = collect([['title' => 'Дом', 'price' => 10000000],
  ...                  ['title' => 'Конура', 'price' => 200]]);
  >>> $el = $coll->firstWhere('price', 200);
  => [ "title" => "Конура", "price" => 200 ]
  >>> $el = $coll->firstWhere('price', 300);
  => null
  >>> $el = $coll->firstWhere('price', '>', 1000000);
  => [ "title" => "Дом", "price" => 10000000 ]
  ```

  Если в вызове методы `firstWhere()` указан только *ключ*, будет возвращен элемент, в котором элемент с заданным *ключом* хранит значение, которое при преобразовании в логический тип дает `true`. Пример:

  ```
  >>> $el = collect([['platform' => 'PHP', 'server' => true],
  ...               ['platform' => 'JavaScript', 'server' => false]])
  ...     ->firstWhere('server');
  => [ "platform" => "PHP", "server" => true ]
  ```

- `firstOrFail()` — в зависимости от формата вызова, возвращает:
  - `firstOrFail()` (без параметров) — первый элемент текущей коллекции. Если текущая коллекция «пуста», возбуждает исключение `Illuminate\Support\ItemNotFoundException`;
  - `firstOrFail(<`*ключ*`>[, <`*оператор сравнения*`>][, <`*значение*`>)` — то же самое, что и метод `firstWhere()`. Если подходящий элемент в текущей коллекции не найден, возбуждает исключение `ItemNotFoundException`;
  - `firstOrFail(<`*анонимная функция*`>)` — первый элемент текущей коллекции, для которого заданная *анонимная функция* вернет значение `true`. *Анонимная функция* должна принимать в качестве параметров значение и ключ очередного элемента коллекции. Если подходящий элемент в текущей коллекции не найден, возбуждает исключение `ItemNotFoundException`;
- `sole()` — аналогичен `firstOrFail()`, только, в зависимости от формата вызова:
  - `sole()` (без параметров) — если в текущей коллекции присутствует более одного элемента, возбуждает исключение `Illuminate\Collections\MultipleItemsFoundException`;
  - `sole(<`*ключ*`>[, <`*оператор сравнения*`>][, <`*значение*`>)`
    `sole(<`*анонимная функция*`>)`

    Если найдено несколько элементов, удовлетворяющих заданным условиям, возбуждает исключение `MultipleItemsFoundException`;
- `where()` — аналогичен `firstWhere()`, только возвращает новую коллекцию, составленную из элементов текущей коллекции, которые удовлетворяют заданным условиям:

```
>>> $coll = collect([['title' => 'Дом', 'price' => 10000000],
...                  ['title' => 'Конура', 'price' => 200],
...                  ['title' => 'Дача', 'price' => 200000],
...                  ['title' => 'Пылесос', 'price' => 1000]]);
>>> $coll2 = $coll->where('title', 'Конура');
=> [ 1 => ["title" => "Конура", "price" => 200] ]
>>> $coll2 = $coll->where('price', '<', 5000);
=> [ 1 => ["title" => "Конура", "price" => 200],
     3 => ["title" => "Пылесос", "price" => 1000] ]
```

  Метод `where()` выполняет обычное (нестрогое) сравнение:

```
>>> $coll2 = $coll->where('price', '<', '5000');
=> [ 1 => ["title" => "Конура", "price" => 200],
     3 => ["title" => "Пылесос", "price" => 1000] ]
```

- `whereStrict()` — то же самое, что и `where()`, только выполняет строгое сравнение;
- `whereBetween(<`*ключ*`>, <`*диапазон*`>)` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элементы с заданным *ключом* хранят значения, укладывающиеся в заданный *диапазон*. *Диапазон* задается в виде массива из двух элементов: начального и конечного значения. Пример:

```
>>> $coll2 = $coll->whereBetween('price', [100000, 50000000]);
=> [ 0 => ["title" => "Дом", "price" => 10000000],
     2 => ["title" => "Дача", "price" => 200000] ]
```

- `whereNotBetween()` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элементы с заданным *ключом* хранят значения, находящиеся за пределами заданного *диапазона*. В остальном аналогичен методу `whereBetween()`. Пример:

```
>>> $coll2 = $coll->whereNotBetween('price', [100000, 50000000]);
=> [ 1 => ["title" => "Конура", "price" => 200],
     3 => ["title" => "Пылесос", "price" => 1000] ]
```

- `whereIn()` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элементы с заданным *ключом* хранят значения, содержащиеся в заданном *массиве*:

```
whereIn(<ключ>, <массив значений>[, <строгое сравнение?>=false])
```

Пример:

```
>>> $coll2 = $coll->whereIn('title', ['Дача', 'Конура']);
=> [ 1 => ["title" => "Конура", "price" => 200],
     2 => ["title" => "Дача", "price" => 200000] ]
```

По умолчанию метод `whereIn()` выполняет обычное (нестрогое) сравнение. Чтобы он выполнял строгое сравнение, необходимо дать параметру *строгое сравнение* значение `true`;

- `whereInStrict(<ключ>, <массив значений>)` — то же самое, что и `whereIn()`, только выполняет строгое сравнение;
- `whereNotIn()` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элементы с заданным *ключом* хранят значения, отсутствующие в заданном *массиве*. В остальном аналогичен методу `wherein()`. Пример:

```
>>> $coll2 = $coll->whereNotIn('title', ['Дача', 'Конура']);
=> [ 0 => ["title" => "Дом", "price" => 10000000],
     3 => ["title" => "Пылесос", "price" => 1000] ]
```

- `whereNotInStrict(<ключ>, <массив значений>)` — то же самое, что и `whereNotIn()`, только выполняет строгое сравнение;
- `whereNull(<ключ>)` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элемент с указанным *ключом* хранит `null`:

```
>>> $coll = collect([['title' => 'Дом', 'content' => 'Большой'],
...                  ['title' => 'Конура', 'content' => null]]);
>>> $coll2 = $coll->whereNull('content');
=> [ 1 => ["title" => "Конура", "content" => null] ]
```

- `whereNotNull(<ключ>)` — возвращает новую коллекцию, составленную из элементов текущей коллекции, в которых элемент с указанным *ключом* хранит значение, отличное от `null`:

```
>>> $coll2 = $coll->whereNotNull('content');
=> [ ["title" => "Дом", "content" => "Большой"] ]
```

- `whereInstanceOf(<путь к классу>)` — возвращает новую коллекцию, составленную из элементов текущей коллекции, которые являются объектами класса с указанным *путем*:

```
>>> use App\Models\Rubric;
>>> use App\Models\Bb;
>>> $coll = collect([new Rubric, new Rubric, new Bb, new Bb]);
>>> $coll2 = $coll->whereInstanceOf(Bb::class);
=> [ 2 => App\Models\Bb, 3 => App\Models\Bb ]
```

- `filter([<анонимная функция>=null])` — возвращает новую коллекцию, составленную из тех элементов текущей коллекции, для которых заданная *анонимная функция* вернет `true`. *Анонимная функция* должна принимать значение и индекс (ключ) очередного элемента текущей коллекции. Пример:

```
>>> $coll = collect([1, 2, 3, 4])->filter(function ($value, $index) {
...     return $value % 2 == 0;
... });
=> [ 1 => 2, 3 => 4 ]
```

  Если *анонимная функция* не указана, будет возвращена коллекция из элементов текущей коллекции, чье значение при преобразовании в логический тип даст `true`:

```
>>> $coll = collect([1, 0, 3, '', [], null])->filter();
=> [ 0 => 1, 2 => 3 ]
```

- `reject([<анонимная функция>=true])` — возвращает новую коллекцию, составленную из тех элементов текущей коллекции, для которых заданная *анонимная функция* вернет `false`. *Анонимная функция* должна принимать значение и индекс (ключ) очередного элемента текущей коллекции. Пример:

```
>>> $coll = collect([1, 2, 3, 4])->reject(function ($value, $index) {
...     return $value % 2 == 0;
... });
=> [ 0 => 1, 2 => 3 ]
```

  Если *анонимная функция* не указана, будет возвращена коллекция из элементов текущей коллекции, чье значение при преобразовании в логический тип даст `false`:

```
>>> $coll = collect([1, 0, 3, '', [], null])->reject();
=> [ 1 => 0, 3 => "", 4 => [], 5 => null ]
```

- `partition()` — возвращает новую коллекцию из двух элементов: коллекции, содержащей элементы текущей коллекции, которые удовлетворяют заданным условиям, и коллекции с элементами, не удовлетворяющими заданным условиям. Формат вызова такой же, как и у метода `firstWhere()`. Пример:

```
>>> [$coll1, $coll2] = collect([1, 2, 3, 4, 5, 6])
...         ->partition(function ($el) { return $el % 2 == 0; });
>>> $coll1;
=> [ 1 => 2, 3 => 4, 5 => 6 ]
>>> $coll2;
=> [ 0 => 1, 2 => 3, 4 => 5 ]
```

```
>>> [$coll1, $coll2] = collect([
...                  ['platform' => 'PHP', 'side' => 'server'],
...                  ['platform' => 'JavaScript', 'side' => 'client'],
...                  ['platform' => 'Python', 'side' => 'server']])
...                   ->partition('side', 'client');
>>> $coll1;
=> [ 1 => ["platform" => "JavaScript", "side" => "client"] ]
>>> $coll2;
=> [ 0 => ["platform" => "PHP", "side" => "server"],
     2 => ["platform" => "Python", "side" => "server"] ]
```

15.1.7. Упорядочивание элементов коллекции

- `sort([<`*анонимная функция*`>=null])` — сортирует элементы текущей индексированной коллекции по возрастанию значений и возвращает результат в виде новой коллекции:

```
>>> $coll = collect([5, -90, 34, 7, 658, 23]);
>>> $coll2 = $coll->sort();
=> [ 1 => -90, 0 => 5, 3 => 7, 5 => 23, 2 => 34, 4 => 658 ]
```

  Если требуется отсортировать элементы в другом порядке, следует указать *анонимную функцию*, которая должна принимать в качестве параметров два элемента коллекции и возвращать любое отрицательное число, если первый элемент меньше второго, 0 — если элементы равны, и любое положительное число, если первый элемент больше второго. Пример:

```
>>> $coll2 = $coll->sort(function ($el1, $el2) {
...     return $el2 - $el1;
... });
=> [ 4 => 658, 2 => 34, 5 => 23, 3 => 7, 0 => 5, 1 => -90 ]
```

- `sortDesc([<`*параметры сортировки*`>=SORT_REGULAR])` — сортирует элементы текущей индексированной коллекции по убыванию значений и возвращает результат в виде новой коллекции. Можно указать любые *параметры сортировки*, поддерживаемые функцией PHP `sort()`. Пример:

```
>>> $coll2 = $coll->sortDesc(SORT_NUMERIC);
=> [ 4 => 658, 2 => 34, 5 => 23, 3 => 7, 0 => 5, 1 => -90 ]
```

- `sortBy()` — сортирует элементы текущей коллекции, хранящей ассоциативные массивы, по возрастанию значений элементов с указанным *ключом* и возвращает результат в виде новой коллекции:

```
sortBy(<ключ>|<анонимная функция>[,
       <параметры сортировки>=SORT_REGULAR[, <по убыванию?>=false]])
```

  Можно указать *параметры сортировки*, поддерживаемые функцией PHP `sort()`. Если дать параметру *по убыванию* значение `true`, сортировка будет выполняться по убыванию значений. Примеры:

```
>>> $coll = collect([['platform' => 'PHP', 'side' => 'server'],
...                  ['platform' => 'JavaScript', 'side' => 'client'],
...                  ['platform' => 'Python', 'side' => 'server']]);
```

```
>>> $coll2 = $coll->sortBy('platform');
=> [ 1 => ["platform" => "JavaScript", "side" => "client"],
     0 => ["platform" => "PHP", "side" => "server"],
     2 => ["platform" => "Python", "side" => "server"] ]
>>> $coll2 = $coll->sortBy('platform', SORT_STRING, true);
=> [ 2 => ["platform" => "Python", "side" => "server"],
     0 => ["platform" => "PHP", "side" => "server"],
     1 => ["platform" => "JavaScript", "side" => "client"] ]
```

Вместо *ключа* можно указать *анонимную функцию*, принимающую очередной элемент коллекции и его индекс (ключ) и возвращающую значение, по которому будет выполняться сортировка:

```
>>> $coll2 = $coll->sortBy(function ($el, $index) {
...     return ($el['side'] == 'server' ? '!' : '') . $el['platform'];
... });
=> [ 0 => ["platform" => "PHP", "side" => "server" ],
     2 => ["platform" => "Python", "side" => "server"],
     1 => ["platform" => "JavaScript", "side" => "client"] ]
```

- `sortByDesc()` — то же самое, что и `sortBy()`, только сортирует элементы по убыванию значений элементов:

```
sortByDesc(<ключ>|<анонимная функция>[, <параметры сортировки>=SORT_REGULAR])
```

- `sortKeys()` — сортирует элементы текущей ассоциативной коллекции по возрастанию ключей элементов и возвращает результат в виде новой коллекции:

```
sortKeys(<параметры сортировки>=SORT_REGULAR[, <по убыванию?>=false]])
```

Можно указать *параметры сортировки*, поддерживаемые функцией PHP `sort()`. Если дать параметру *по убыванию* значение `true`, сортировка будет выполняться по убыванию ключей. Пример:

```
>>> $coll = collect(['PHP' => '8.1', 'Laravel' => '9.9',
...                                  'Node' => '18.00']);
>>> $coll2 = $coll->sortKeys();
=> [ "Laravel" => "9.9", "Node" => "18.00", "PHP" => "8.1" ]
```

- `sortKeysDesc([<параметры сортировки>=SORT_REGULAR])` — сортирует элементы текущей ассоциативной коллекции по убыванию ключей и возвращает результат в виде новой коллекции. Можно указать *параметры сортировки*, поддерживаемые функцией PHP `sort()`. Пример:

```
>>> $coll2 = $coll->sortKeysDesc();
=> [ "PHP" => "8.1", "Node" => "18.00", "Laravel" => "9.9" ]
```

- `sortKeysUsing(<имя функции>)` — сортирует элементы текущей ассоциативной коллекции по возрастанию ключей, используя для их сравнения функцию с заданным в виде строки *именем*. Эта функция должна удовлетворять требованиям, предъявляемым функцией PHP `uksort()` (которая и используется для сортировки). Пример:

```
>>> $coll = collect(['php' => '8.1', 'Laravel' => '9.9',
...                                  'node' => '18.00']);
>>> $coll2 = $coll->sortKeysUsing('strnatcasecmp');
=> [ "Laravel" => "9.9", "node" => "18.00", "php" => "8.1" ]
```

- `reverse()` — меняет порядок следования элементов в текущей коллекции на обратный, сохраняя ключи. Результат возвращает в виде новой коллекции. Пример:

```
>>> $coll = collect([1, 2, 3, 4, 5])->reverse();
=> [ 4 => 5, 3 => 4, 2 => 3, 1 => 2, 0 => 1 ]
```

- `shuffle([`*<значение переинициализации>*`=null])` — случайным образом перемешивает элементы текущей коллекции и возвращает результат в виде новой коллекции. Можно указать *значение переинициализации* для встроенного в PHP генератора псевдослучайных чисел.

15.1.8. Группировка элементов коллекций

- `groupBy()` — группирует элементы текущей ассоциативной коллекции согласно заданному *критерию* или критериям, содержащимся в *массиве*, и возвращает результат в виде новой коллекции. Для каждого заданного критерия группировки в результирующей коллекции создается элемент с ключом, совпадающим с критерием, который хранит массив элементов, входящих в соответствующую группу. Формат вызова:

```
groupBy(<критерий>|<массив критериев>[, <сохранять ключи?>=false])
```

Примеры:

```
>>> $coll = collect([
...         ['language' => 'JavaScript', 'framework' => 'Express',
...          'server' => true],
...         ['language' => 'JavaScript', 'framework' => 'Vue',
...          'server' => false],
...         ['language' => 'JavaScript', 'framework' => 'KOA',
...          'server' => true],
...         ['language' => 'PHP', 'framework' => 'Laravel',
...          'server' => true]]);
>>> $coll1 = $coll->groupBy('language');
=> [ "JavaScript" => [["language" => "JavaScript",
                       "framework" => "Express", "server" => true],
                      ["language" => "JavaScript",
                       "framework" => "Vue", "server" => false],
                      ["language" => "JavaScript",
                       "framework" => "KOA", "server" => true]],
     "PHP"        => [["language" => "PHP",
                       "framework" => "Laravel", "server" => true]]]
>>> $coll1 = $coll->groupBy(['language', 'server']);
=> [ "JavaScript" => [1 => [["language" => "JavaScript",
                             "framework" => "Express",
                             "server" => true],
                            ["language" => "JavaScript",
                             "framework" => "KOA",
                             "server" => true]],
                      0 => [["language" => "JavaScript",
                             "framework" => "Vue",
                             "server" => false]]],
```

```
    "PHP"        => [1 => [["language" => "PHP",
                           "framework" => "Laravel",
                           "server" => true]]] ]
```

В качестве *критерия* можно передать анонимную функцию, принимающую значение очередного элемента текущей коллекции и его индекс (ключ) и возвращающую значение, которое станет ключом:

```
>>> $coll1 = $coll->groupBy(['language', function ($el, $index) {
...     return $el['server'] ? 'server' : 'client';
... }]);
=> [ "JavaScript" => ['server' => [["language" => "JavaScript",
                                   "framework" => "Express",
                                   "server" => true],
                                  ["language" => "JavaScript",
                                   "framework" => "KOA",
                                   "server" => true]],
                      'client' => [["language" => "JavaScript",
                                   "framework" => "Vue",
                                   "server" => false]]],
     "PHP"        => ['server' => [["language" => "PHP",
                                   "framework" => "Laravel",
                                   "server" => true]]]]
```

Если дать параметру *сохранять ключи* значение `true`, метод будет сохранять ключи элементов вложенных массивов:

```
>>> $coll1 = $coll->groupBy('language', true);
=> [ "JavaScript" => [0 => [ ... ],
                      1 => [ ... ],
                      2 => [ ... ]],
     "PHP"        => [3 => [ ... ]]]
```

- `mapToGroups(<`*анонимная функция*`>)` — группирует текущую коллекцию, хранящую ассоциативные массивы, на основании результата, возвращенного заданной *анонимной функцией*. Последняя должна принимать очередной элемент коллекции и его индекс (ключ) и возвращать ассоциативный массив. Пример:

```
>>> $coll = collect([['platform' => 'PHP', 'side' => 'server'],
...                  ['platform' => 'JavaScript', 'side' => 'client'],
...                  ['platform' => 'Python', 'side' => 'server']])
...       ->mapToGroups(function ($el, $index) {
...             return [$el['side'] => $el['platform']];
...         });
=> [ "server" => ["PHP", "Python"], "client" => ["JavaScript"] ]
```

15.1.9. Агрегатные вычисления в коллекциях

- `avg([<`*ключ*`>|<`*анонимная функция*`>=null])` — возвращает среднее арифметическое, вычисленное на основе числовых элементов текущей коллекции:

```
>>> echo collect([1, 2, 3, 4])->avg();
2.5
```

Если текущая коллекция хранит многомерный массив, следует указать *ключ* элементов вложенных массивов, на основе значений которых будет рассчитываться среднее:

```
>>> $coll = collect([['num' => 1], ['num' => 2], ['num' => 3],
...                                               ['num' => 4]]);
>>> echo $coll->avg('num');
2.5
```

Также можно указать *анонимную функцию*, принимающую в качестве параметра очередной элемент коллекции и возвращающую очередное значение, участвующее в вычислении среднего:

```
>>> echo $coll->avg(function ($el) { return $el['num']; });
2.5
```

- `average()` — то же самое, что и `avg()`;
- `sum()` — возвращает сумму всех числовых значений текущей коллекции. Формат вызова такой же, как и у метода `avg()`. Пример:

```
>>> echo collect([1, 2, 3, 4])->sum();
10
```

- `min()` — возвращает минимальное из всех числовых значений текущей коллекции. Формат вызова такой же, как и у метода `avg()`. Пример:

```
>>> echo collect([78, 9, -376, 52])->max();
-376
```

- `max()` — возвращает максимальное из всех числовых значений текущей коллекции. Формат вызова такой же, как и у метода `avg()`. Пример:

```
>>> echo collect([78, 9, -376, 52])->max();
78
```

- `median()` — возвращает медиану, вычисленную на основе числовых значений текущей коллекции. Формат вызова такой же, как и у метода `avg()`;
- `mode()` — возвращает моду, вычисленную на основе числовых значений текущей коллекции. Формат вызова такой же, как и у метода `avg()`;
- `pipe(<`*анонимная функция*`>)` — вызывает указанную *анонимную функцию*, передавая ей саму текущую коллекцию, и выводит возвращенный этой *функцией* результат:

```
>>> echo collect([1, 2, 3, 4])
...      ->pipe(function ($coll) { return $coll->avg(); });
2.5
```

15.1.10. Получение сведений о коллекции

- `count()` — возвращает размер текущей коллекции;
- `isEmpty()` — возвращает `true`, если текущая коллекция «пуста», и `false` — в противном случае;
- `isNotEmpty()` — возвращает `true`, если текущая коллекция не «пуста», и `false` — в противном случае;

- `countBy([<анонимная функция>=null])` — возвращает количество элементов текущей коллекции, хранящих одинаковые значения, в виде ассоциативного массива, ключами элементов которого являются значения элементов в текущей коллекции, а значениями — количества этих элементов:

```
>>> $coll = collect([1, 3, 1, 3, 3, 70])->countBy();
=> [ 1 => 2, 3 => 3, 70 => 1 ]
```

  Можно указать *анонимную функцию*, принимающую значение очередного элемента текущей коллекции и возвращающую результат, который станет ключом элемента ассоциативного массива, выдаваемого методом `countBy()`. Пример:

```
>>> $coll = collect([1, 3, 1, 3, 3, 70])->countBy(function ($value) {
...     return ($value > 10) ? '>10' : '<=10';
... });
=> [ "<=10" => 5, ">10" => 1 ]
```

15.1.11. Прочие инструменты для обработки коллекций

- `toArray()` — преобразует текущую коллекцию в обычный массив PHP и возвращает его;
- `toJson()` — преобразует текущую коллекцию в формат JSON и возвращает преобразованные данные в виде строки;
- `all()` — возвращает массив, на основе которого была создана коллекция;
- `unwrap(<значение>)` — статический метод. Если заданное *значение* является коллекцией, возвращает массив, на основе которого она была создана. В противном случае возвращается само *значение*;
- `implode([<ключ>, ]<разделитель>)` — объединяет элементы текущей коллекции в строку, разделяя их заданным *разделителем*, и возвращает ее в качестве результата:

```
>>> echo collect([1, 2, 3, 4, 5])->implode(', ');
1, 2, 3, 4, 5
```

  Если текущая коллекция содержит ассоциативные массивы, следует указать *ключ* элементов, значения которых будет объединяться в строку:

```
>>> echo collect([['p' => 'PHP', 'fr' => 'Laravel'],
...               ['p' => 'JavaScript', 'fr' => 'AngularJS']])
...    ->implode('p', ', ');
PHP, JavaScript
```

- `join(<разделитель>[, <последний разделитель>=''])` — объединяет элементы текущей коллекции в строку, разделяя их заданным *разделителем*, и возвращает ее в качестве результата. *Последний разделитель* отделит друг от друга предпоследний и последний элементы. Если он не указан, для этого будет использован *разделитель*. Примеры:

```
>>> echo collect([1, 2, 3, 4])->join(', ');
1, 2, 3, 4
>>> echo collect([1, 2, 3, 4])->join(', ', ' и ');
1, 2, 3 и 4
```

- `collapse()` — уменьшает мерность текущей коллекции на единицу (например, двумерную коллекцию превращает в одномерную) и возвращает полученный результат в виде новой коллекции:

```
>>> $coll = collect([[1, 2], [3, 4, 5]])->collapse();
=> [ 1, 2, 3, 4, 5 ]
>>> $coll = collect([[1, 2], [3, [4, 5]]])->collapse();
=> [ 1, 2, 3, [4, 5] ]
```

- `flatten([<`*уровень*`>=INF])` — уменьшает мерность текущей коллекции на заданный *уровень* (по умолчанию: «бесконечность», т. е. до одномерной коллекции) и возвращает полученный результат в виде новой коллекции:

```
>>> $coll = collect([1, [2, 3, [4, 5, [6, 7]]]])->flatten();
=> [ 1, 2, 3, 4, 5, 6, 7 ]
>>> $coll = collect([1, [2, 3, [4, 5, [6, 7]]]])->flatten(2);
=> [ 1, 2, 3, 4, 5, [6, 7] ]
```

- `keyBy(<`*ключ*`>|<`*анонимная функция*`>)` — при вызове у коллекции, хранящей вложенные ассоциативные массивы, задает у каждого элемента ключ, взятый из элемента вложенных массивов с указанным *ключом*, и возвращает результат в виде новой коллекции:

```
>>> $coll = collect([['p' => 'PHP', 'fr' => 'Laravel'],
...                  ['p' => 'JavaScript', 'fr' => 'AngularJS']])
...       ->keyBy('p');
=> [ "PHP"        => ["p" => "PHP", "fr" => "Laravel"],
     "JavaScript" => ["p" => "JavaScript", "fr" => "AngularJS"] ]
```

  Также можно указать *анонимную функцию*, принимающую с параметром очередной элемент текущей коллекции и возвращающую строку, которая станет ключом этого элемента:

```
>>> use \Illuminate\Support\Str;
>>> $coll = collect([['p' => 'PHP', 'fr' => 'Laravel'],
...                  ['p' => 'JavaScript', 'fr' => 'AngularJS']])
...       ->keyBy(function ($el) { return Str::lower($el['p']); });
=> [ "php"        => ["p" => "PHP", "fr" => "Laravel"],
     "javascript" => ["p" => "JavaScript", "fr" => "AngularJS"] ]
```

- `map(<`*анонимная функция*`>)` — перебирает элементы текущей коллекции, для каждого вызывает заданную *анонимную функцию*, добавляет возвращаемые ей результаты в новую коллекцию, которую и возвращает в качестве результата. *Анонимная функция* должна принимать значение очередного элемента коллекции и его индекс (ключ). Пример:

```
>>> $coll = collect([1, 2, 3, 4])->map(function ($value, $index) {
...     return $value ** 2;
... });
=> [ 1, 4, 9, 16 ]
```

- `flatMap(<`*анонимная функция*`>)` — аналогичен `map()`, только дополнительно уменьшает мерность возвращаемой коллекции на единицу:

```
>>> $coll = collect([['platform' => 'PHP'], ['database' => 'MySQL'],
...                  ['framework' => 'Laravel']])
...       ->flatMap(function ($el) {
...             return array_map('strtolower', $el);
...       });
=> ["platform" => "php", "database" => "mysql",
    "framework" => "laravel" ]
```

- `mapSpread(<анонимная функция>)` — перебирает текущую коллекцию, хранящую вложенные массивы, для каждого элемента вызывает заданную *анонимную функцию*, передавая ей элементы вложенного массива и его индекс (ключ), добавляет возвращаемые ею результаты в новую коллекцию, которую и возвращает:

```
>>> $coll = collect([[1, 2], [3, 4], [4, 5]])
...       ->mapSpread(function($el1, $el2, $index) {
...           return $el1 * $el2;
...         });
=> [ 2, 12, 20 ]
```

- `mapWithKeys(<анонимная функция>)` — перебирает текущую коллекцию, хранящую вложенные ассоциативные массивы, для каждого элемента вызывает заданную *анонимную функцию*, передавая ей очередной элемент и его индекс (ключ), добавляет возвращаемые ею результаты в новую коллекцию, которую и возвращает:

```
>>> $coll = collect([['platform' => 'PHP', 'side' => 'server'],
...                  ['platform' => 'JavaScript', 'side' => 'client'],
...                  ['platform' => 'Python', 'side' => 'server']])
...       ->mapWithKeys(function ($el, $index) {
...           return [$el['platform'] => $el['side']];
... });
=> [ "PHP" => "server", "JavaScript" => "client",
     "Python" => "server" ]
```

- `mapInto(<путь к классу>)` — перебирает текущую коллекцию, для каждого элемента создает объект класса с указанным *путем*, передавая значение этого элемента конструктору класса, и возвращает новую коллекцию, составленную из полученных объектов:

```
>>> class Platform {
...     public function __construct($name) {
...         $this->name = $name;
...     }
... }
>>> $coll = collect(['PHP', 'JavaScript', 'Python'])
...        ->mapInto(Platform::class);
=> [ Platform {#3094 +"name": "PHP"},
     Platform {#3093 +"name": "JavaScript"},
     Platform {#3092 +"name": "Python"} ]
```

- `transform(<анонимная функция>)` — перебирает элементы текущей коллекции, для каждого вызывает заданную *анонимную функцию*, заменяет значение этого элемента возвращенным ею результатом и возвращает текущую коллекцию. *Анонимная функ-*

ция должна принимать значение очередного элемента коллекции и его индекс (ключ):

```
>>> $coll = collect([1, 2, 3, 4])
...       ->transform(function ($value, $index) {
...             return $value * $index;
...         });
=> [ 0, 2, 6, 12 ]
```

- `crossJoin(<`*коллекция 1*`>, <`*коллекция 2*`>, ... <`*коллекция n*`>)` — возвращает декартово произведение текущей и всех заданных *коллекций*;
- `reduce(<`*анонимная функция*`>[, <`*изначальное значение*`>=null])` — для каждого элемента текущей коллекции вызывает заданную *анонимную функцию*, после чего удаляет этот элемент. *Анонимная функция* должна принимать два параметра: результат, возвращенный предыдущим вызовом этой же функции, или заданное *изначальное значение* (если это ее первый вызов) и значение очередного элемента текущей коллекции. Когда текущая коллекция «опустеет», возвращает результат последнего вызова *анонимной функции*. Пример:

```
>>> echo collect([1, 2, 3, 4, 5])
...      ->reduce(function ($carry, $el) {
...                   return $carry + $el ** 2;
...               }, 0);
55
```

- `flip()` — в текущей ассоциативной коллекции меняет местами ключи и значения элементов и возвращает результат в виде новой коллекции:

```
>>> $coll = collect(['platform' => 'PHP', 'database' => 'MySQL'])
...         ->flip();
=> [ "PHP" => "platform", "MySQL" => "database" ]
```

- `undot()` — превращает текущую одномерную ассоциативную коллекцию, в которой ключи элементов состоят из нескольких слов, разделенных точками, в многомерную, которую и возвращает:

```
>>> $coll = collect(["backend.platform.base" => "PHP",
...                  "backend.platform.framework" => "Laravel",
...                  "backend.database" => "MySQL",
...                  "frontend" => "Node"])->undot();
=> [ "backend" =>
         ["platform" => ["base" => "PHP", "framework" => "Laravel"],
          "database" => "MySQL"],
     "frontend" => "Node" ]
```

- `tap(<`*анонимная функция*`>)` — вызывает заданную *анонимную функцию*, передавая ей копию текущей коллекции, и возвращает эту копию в качестве результата;
- `when()` — вызывает заданную *анонимную функцию 1*, если указанное *значение* при преобразовании в логический тип дает `true`, и возвращает результат, возвращенный этой функцией. В противном случае вызывает *анонимную функцию 2*, если она указана, и также возвращает выданный ею результат. Если *анонимная функция 2* не указана,

возвращает текущую коллекцию. Обе *анонимные функции* должны принимать два параметра: текущую коллекцию и *значение*. Формат вызова:

```
when(<значение>, <анонимная функция 1>[, <анонимная функция 2>=null])
```

Пример:

```
>>> $coll1 = collect([]);
>>> $coll2 = $coll1->when($coll1->isEmpty(), function($coll, $flag) {
...     return $coll->add(1);
... });
=> [ 1 ]
```

- `unless()` — аналогичен `when()`, только вызывает заданную *анонимную функцию 1*, если указанное *значение* при преобразовании в логический тип дает `false`;
- `whenEmpty()` — вызывает заданную *анонимную функцию 1*, если текущая коллекция «пуста», и возвращает результат, возвращенный этой функцией. В противном случае вызывает *анонимную функцию 2*, если она указана, и также возвращает выданный ею результат. Если *анонимная функция 2* не указана, возвращает текущую коллекцию. Обе *анонимные функции* должны принимать два параметра: текущую коллекцию и логическое значение `true`, если текущая коллекция «пуста», и `false` — в противном случае. Формат вызова:

  ```
  whenEmpty(<анонимная функция 1>[, <анонимная функция 2>=null])
  ```

  Пример:

  ```
  >>> $coll1 = collect([]);
  >>> $coll2 = $coll1->whenEmpty(function($coll, $flag) {
  ...     return $coll->add(1);
  ... });
  => [ 1 ]
  ```

- `unlessNotEmpty()` — то же самое, что и `whenEmpty()`;
- `whenNotEmpty()` — аналогичен `whenEmpty()`, только вызывает заданную *анонимную функцию 1*, если текущая коллекция не «пуста»;
- `unlessEmpty()` — то же самое, что и `whenNotEmpty()`;
- `zip(<коллекция>)` — возвращает новую двумерную коллекцию, каждый элемент которой представляет собой вложенный массив с двумя элементами, взятыми из текущей и заданной коллекций (вместо нее также можно указать массив):

  ```
  >>> $coll = collect([1, 2, 3])->zip(collect([10, 20, 30]));
  => [ [1, 10], [2, 20], [3, 30] ]
  ```

- `sliding([<размер чанка>=2[, <сдвиг>=1]])` — разбивает текущую коллекцию на набор чанков заданного *размера*, каждый из которых содержит набор последовательно расположенных элементов текущей коллекции, смещенных относительно предыдущего чанка на указанную величину *сдвига*, и также является коллекцией, объединяет чанки в новую коллекцию и возвращает в качестве результата:

  ```
  >>> collect([1, 2, 3, 4, 5])->sliding();
  => [ [1, 2], [1 => 2, 2 => 3], [2 => 3, 3 => 4], [3 => 4, 4 => 5] ]
  ```

```
>>> collect([1, 2, 3, 4, 5])->sliding(3);
=> [ [1, 2, 3], [1 => 2, 2 => 3, 3 => 4], [2 => 3, 3 => 4, 4 => 5] ]
>>> collect([1, 2, 3, 4, 5])->sliding(3, 2);
=> [ [1, 2, 3], [2 => 3, 3 => 4, 4 => 5] ]
```

15.2. Коллекции, заполняемые по запросу

Коллекция, заполняемая по запросу, не хранит в оперативной памяти все содержащиеся в ней элементы, а генерирует их один за другим, как только в них возникает необходимость. Благодаря этому экономится память при работе с коллекциями, содержащими много элементов.

Коллекция, заполняемая по запросу, представляется объектом класса `Illuminate\Support\LazyCollection`.

15.2.1. Создание коллекций, заполняемых по запросу

Создать коллекцию, заполняемую по запросу, можно вызовом конструктора класса `LazyCollection` в следующем формате:

```
LazyCollection(<анонимная фукнция-генератор>)
```

Задаваемая *анонимная функция-генератор* будет генерировать элементы коллекции. Она не должна принимать параметров и должна при каждом вызове возвращать очередной элемент. Пример:

```
>>> use Illuminate\Support\LazyCollection;
>>> $coll = new LazyCollection(function() {
...     for ($i = 0; $i <= 1000; $i++)
...         yield $i;
... });
>>> echo $coll->sum();
500500
```

Можно использовать статический метод `make()` класса `LazyCollection`, вызываемый в таком же формате:

```
>>> $coll = LazyCollection::make(function() { ... });
```

Также поддерживаются методы `times()` и `range()`.

Метод `lazy()` класса `Collection` создает на основе текущей коллекции и возвращает в качестве результата коллекцию, заполняемую по запросу:

```
$col1 = collect([ ... ])->lazy();
```

15.2.2. Работа с коллекциями, заполняемыми по запросу

Класс `LazyCollection` поддерживает те же методы, что и класс `Collection`, за исключением: `add()`, `doesntContain()`, `forget()`, `lazy()`, `pop()`, `prepend()`, `pull()`, `push()`, `put()`, `range()`, `shift()`, `sliding()`, `skipUntil()`, `skipWhile()`, `sortDesc()`, `sortKeysUsing()`, `splice()`, `splitIn()`, `takeUntil()`, `takeWhile()`, `transform()`, `undot()`, `value()`, `whereNotNull()` и `whereNull()`. Также поддерживаются три специфических метода:

- `takeUntilTimeout(<временна́я отметка>)` — возвращает новую коллекцию, которую можно перебирать лишь до наступления момента времени, заданного переданной методу *временно́й отметкой*. При наступлении заданного момента времени коллекция перестанет перебираться. *Временну́ю отметку* можно задать в виде значения типа PHP `DateTimeInterface` или объекта класса `Carbon`. Пример:

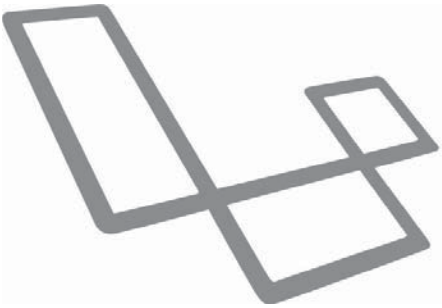
```
$coll = LazyCollection::times(INF)
                      ->takeUntilTimeout(now()->addMinute());
// Эту коллекцию можно перебирать лишь в течение минуты
$coll->each(function ($number) { ... });
// Все, минута прошла, поехали дальше…
```

- `tapEach(<анонимная функция>)` — аналогичен `each()` (см. *разд. 15.1.5*), только выполняется не сразу, а как только возникнет необходимость в результатах его работы:

```
>>> // Сейчас метод tapEach() не вызывается, поскольку
>>> // результаты его работы еще не запрошены
>>> $coll = LazyCollection::times(10)->tapEach(function ($el) {
...     echo $el . ' | ';
... });
>>> // А вот теперь он вызывается
>>> $coll->all();
1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
```

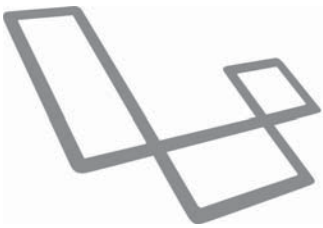
- `remember()` — возвращает новую коллекцию, заполняемую по запросу, которая будет кешировать в памяти все сгенерированные элементы с тем, чтобы не генерировать их повторно (в некоторых случаях это позволит повысить производительность за счет несколько большего расхода памяти):

```
>>> $coll = LazyCollection::times(10)->remember();
>>> // Первые 5 элементов коллекции будут сгенерированы и кешированы
>>> // в памяти
>>> $coll->take(5);
>>> // Первые 5 кешированных элементов повторно генерироваться
>>> // не будут, оставшиеся 5 будут сгенерированы и также кешированы
>>> $coll->take(10);
```



ЧАСТЬ III

Расширенные инструменты и дополнительные библиотеки



ГЛАВА 16

Базы данных и модели: расширенные инструменты

16.1. Отложенная и немедленная выборка связанных записей

По умолчанию Laravel реализует *отложенную выборку связанных записей*, при которой связанные записи первичной модели загружаются из базы данных только тогда, когда выполняется обращение к ним. Пример:

```
>>> use App\Models\Bb;
>>> // Запрашиваем перечень объявлений. Будут загружены лишь объявления,
>>> // но не связанные с ними рубрики.
>>> $bbs = Bb::all();
>>> $bb = $bbs[0];
>>> // Запрашиваем название рубрики, связанной с первым объявлением.
>>> // Именно в этот момент и будет загружена связанная рубрика.
>>> $rubricName = $bb->rubric->name;
```

В большинстве случаев это оправданно, поскольку неизвестно, будут ли запрашиваться связанные записи. Однако иногда имеет смысл выполнить одновременную выборку и записей вторичной таблицы, и связанных с ними записей первичной таблицы — чтобы повысить производительность. То есть реализовать *немедленную выборку связанных записей*.

Немедленную выборку можно реализовать:

- на уровне текущего запроса — вызвав метод `with()`, поддерживаемый моделями и построителем запросов и имеющий два формата вызова:

  ```
  with(<связь 1>, <связь 2>, ... <связь n>)
  with(<массив связей>)
  ```

  В качестве отдельной *связи* можно указать:

  - имя связи с первичной моделью в виде строки:

    ```
    >>> // Извлекаем, помимо объявлений, связанные с ними рубрики
    >>> // второго уровня
    >>> $bbs = Bb::with('rubric')->get();
    ```

- строку, отражающую иерархию связанных друг с другом моделей, в которой имена связей с первичными моделями разделены точками:

```
>>> // Дополнительно извлекаем рубрики первого уровня, связанные
>>> // с рубриками второго уровня
>>> $bbs = Bb::with('rubric', 'rubric.parent')->get();
```

- строку формата *<имя связи>:<список полей через запятую>* — для выборки из связанных записей только полей с имеющимися в *списке* именами. В *списке* полей обязательно должны присутствовать ключевое поле и, при выборке записей из иерархически связанной таблицы, поле внешнего ключа:

```
>>> $bbs = Bb::with('rubric:id,name')->get();

>>> $bbs = Bb::with(['rubric:id,name,parent_id',
...                  'rubric.parent:id,name'])->get();
```

❐ на уровне текущей записи, уже выбранной из базы, — вызвав метод `load()` модели, имеющий тот же формат вызова, что и метод `with()`:

```
>>> $bb = Bb::find(1);
>>> $bb->load('rubric', 'rubric.parent');
```

Метод `loadMissing()` аналогичен методу `load()`, но выполняет немедленную загрузку записей только в том случае, если они не были загружены ранее;

❐ на уровне модели — в таком случае немедленная выборка связанных записей будет выполняться всегда. Массив имен связей с первичными моделями следует занести в защищенное свойство `with` класса модели. Пример:

```
class Bb extends Model {
    protected $with = ['rubric', 'rubric.parent'];
    . . .
}
```

После чего можно выполнять выборку записей без применения метода `with()`:

```
>>> $bbs = Bb::all();
```

Если требуется на уровне текущего запроса отключить немедленную выборку у каких-либо связей, приведенных в свойстве `with`, следует указать эти связи в вызове метода `without()`, поддерживаемого построителем запросов и имеющего тот же формат вызова, что и `with()`. Пример:

```
>>> // Отключаем на уровне текущего запроса выборку связанных рубрик
>>> // первого уровня. Будут выбраны лишь рубрики второго уровня.
>>> $bbs = Bb::without('rubric.parent')->get();
```

Есть возможность на уровне текущего запроса перекрыть указания, сделанные в свойстве `with` класса модели, и задать свои связи для немедленной выборки. Для этого следует использовать метод `withOnly()` построителя запросов, имеющий тот же формат вызова, что и `with()`. Пример:

```
>>> // Указываем на уровне текущего запроса выполнить выборку только
>>> // связанных рубрик второго уровня
>>> $bbs = Bb::withOnly('rubric')->get();
```

Также поддерживается немедленная выборка связанных записей вторичной таблицы при выборке записей первичной таблицы. Ее можно выполнить как на уровне текущего запроса:

```
>>> use App\Models\Rubric;
>>> // Извлекаем, помимо рубрик, связанные к ними объявления
>>> $rubrics = Rubric::with('bbs')->get();
```

так и на уровне текущей записи:

```
>>> $rubric = Rubric::find(11);
>>> // Извлекаем связанные с этой рубрикой объявления и пользователей,
>>> // связанных с каждым из извлеченных объявлений
>>> $rubric->loadMissing(['bbs', 'bbs.user']);
```

Можно отсортировать записи вторичной таблицы, подвергаемые немедленной выборке, а также указать у них условия фильтрации. В последнем случае будут немедленно выбраны только записи, удовлетворяющие заданным условиям, а остальные будут выбраны позже, в результате отложенной выборки. Чтобы реализовать это, в массив связей, передаваемый методам `with()`, `load()`, `loadMissing()` или `withOnly()`, следует вставить элемент формата *<имя связи> => <анонимная функция>*. Указываемая *анонимная функция* должна принимать объект построителя запросов и с его помощью задавать необходимые условия фильтрации и сортировки. Пример:

```
>>> // Выполняем немедленную выборку только объявлений с заявленной ценой
>>> // более 1 млн руб.
>>> $rubrics = Rubric::with(['bbs' => function ($query) {
...     $query->where('price', '>', 1000000);
... }])->get();
```

16.2. Выборка наборов записей по частям

Методы `all()` и `get()` построителя запросов извлекают из базы данных все записи, удовлетворяющие заданным условиям фильтрации. Если извлеченный набор записей слишком велик, содержащая его коллекция займет большой объем оперативной памяти, что снизит производительность.

Решить эту проблему можно, выбирая наборы записей по частям (*чанкам*). Коллекция, содержащая чанк, отнимет меньше памяти, чем коллекция с полным набором записей.

Для выборки наборов записей по частям применяются следующие методы, поддерживаемые построителем запросов:

- `chunk(<размер чанка>, <анонимная функция>)` — выбирает набор записей по чанкам указанного *размера*, сохраняет каждый чанк в обычной коллекции `Collection` (см. *разд. 15.1*) и вызывает заданную *анонимную функцию*. Последняя должна принимать в качестве параметров коллекцию, содержащую очередной чанк, и порядковый номер этого чанка (нумерация чанков начинается с 1). Если *анонимная функция* вернет `false`, обработка набора записей будет прервана. Метод возвращает `true`, если набор записей был обработан до конца, и `false` — если обработка была прервана возвратом из *анонимной функции* значения `false`. Пример:

```
>>> $bbs = Bb::chunk(5, function ($part, $index) {
...     echo 'Часть №', $index, ":\r\n";
...     foreach ($part as $item)
...         echo $item->title, ' (', $item->content, '): ',
...             $item->price, "\r\n";
... });
Часть №1:
Гараж (На две машины): 300100
Дача (Новая): 100000
Дом (Большой, трехэтажный): 5000000
. . .
Часть №2:
ЗИЛ (Старый, ржавый): 1000000
Запорожец (Старый, ржавый, сильно битый): 10000
. . .
```

- `each(`*<анонимная функция>*`[, `*<размер чанка>*`=1000])` — выбирает набор записей по чанкам указанного *размера* и для каждой записи (не чанка!) вызывает заданную *анонимную функцию*. Последняя должна принимать два параметра: очередную запись в виде объекта модели и порядковый номер этой записи (нумерация записей начинается с 0). Если *анонимная функция* вернет `false`, обработка набора записей будет прервана. Метод возвращает `true`, если набор записей был обработан до конца, и `false` — если обработка была прервана возвратом из *анонимной функции* значения `false`. Пример:

```
>>> $bbs = Bb::each(function ($item, $index) {
...     echo $index + 1, '. ', $item->title, ' (', $item->content,
...         '): ', $item->price, "\r\n";
... }, 5);
1. Гараж (На две машины): 300100
2. Дача (Новая): 100000
3. Дом (Большой, трехэтажный): 5000000
. . .
```

- `lazy([`*<размер чанка>*`=1000])` — возвращает коллекцию, заполняемую по запросу (`LazyCollection`, см. *разд. 15.2*), которая содержит весь набор записей. «За кулисами» выбирает набор записей по чанкам указанного *размера*. Вызывается вместо метода `get()` или `all()`. Пример:

```
>>> foreach (Bb::lazy(5) as $item) {
...     echo $item->title, ' (', $item->content, '): ',
...         $item->price, "\r\n";
... }
```

Является альтернативой методу `each()`, в ряде случаев — более удобной.

Следующие три метода применяются в том случае, если перед выборкой записей производится их фильтрация по какому-либо полю, а в теле заданной *анонимной функции* у выбранных записей исправляется значение того же самого поля, по которому эти записи фильтровались. Эти методы разбивают выбираемый набор записей на чанки, сравнивая ключи записей;

- `chunkById()` — аналог метода `chunk()`:

```
chunkById(<размер чанка>, <анонимная функция>[, <имя ключевого поля>=null])
```

Если *имя ключевого поля* не задано, будет использовано ключевое поле, указанное в классе модели. Пример:

```
>>> $bbs = Bb::where('publish', false)
...      ->chunkById(5, function ($pt, $ind) {
...           foreach ($pt as $item)
...               $item->update(['publish' => true]);
...        }
... );
```

- `eachById()` — аналог метода `each()`:

```
eachById(<анонимная функция>[, <размер чанка>=1000[,
                              <имя ключевого поля>=null]])
```

Если *имя ключевого поля* не задано, будет использовано ключевое поле, указанное в классе модели. Пример:

```
>>> $bbs = Bb::where('publish', false)
...      ->eachById(function ($item, $index) {
...            $item->update(['publish' => true]);
...        }, 5);
```

- `lazyById()` — аналог метода `lazy()`:

```
lazyById([<размер чанка>=100[, <имя ключевого поля>=null]])
```

Если *имя ключевого поля* не задано, будет использовано ключевое поле, указанное в классе модели. Пример:

```
>>> foreach (Bb::where('publish', false)->lazyById(5) as $item) {
...     $item->update(['publish' => true]);
... }
```

Альтернативой всем этим методам является метод `cursor()`, который выбирает сразу весь набор записей, но возвращает его в виде коллекции, заполняемой по запросу:

```
>>> foreach (Bb::latest()->cursor() as $item) { ... }
```

16.3. Фильтрующие связи «многие-со-многими»

Можно создать связь типа «многие-со-многими», которая выдает не все связанные записи, а лишь те, которые удовлетворяют заданным условиям фильтрации *по значениям полей связующей таблицы* (*фильтрующая связь*). Для этого в методе модели, создающем связь, у объекта связи следует вызвать один из методов, приведенных далее. Эти методы аналогичны методам, которые описаны в *разд. 7.3.4*, за тем исключением, что выполняют фильтрацию связанных записей по полям связующей таблицы:

- `wherePivot()` — аналогичен первому формату вызова метода `where()`:

```
class Machine extends Model {
    . . .
```

```
        // Создаем связь, выдающую только те детали, что присутствуют
        // в количестве более одной штуки
        public function sparesMoreThenOne() {
            return $this->belongsToMany(Spare::class)->withPivot('cnt')
                        ->wherePivot('cnt', '>', 1);
        }
    }
    . . .
    >>> use App\Models\Machine;
    >>> $machine = Machine::find(3);
    >>> $spares = $machine->spares;
    >>> foreach ($spares as $spare) {
    ...     echo $spare->name, ': ', $spare->pivot->cnt, "\r\n";
    ... }
    Заклепка: 1
    Шайба: 200
    >>> $spares = $machine->sparesMoreThenOne;
    . . .
    Шайба: 200
```

- `orWherePivot()` — аналогичен первому формату вызова метода `orWhere()`;
- `wherePivotBetween()` — аналогичен методу `whereBetween()`;
- `wherePivotNotBetween()` — аналогичен методу `whereNotBetween()`;
- `orWherePivotBetween()` — аналогичен методу `orWhereBetween()`;
- `orWherePivotNotBetween()` — аналогичен методу `orWhereNotBetween()`;
- `wherePivotIn()` — аналогичен методу `whereIn()`;
- `wherePivotNotIn()` — аналогичен методу `whereNotIn()`;
- `orWherePivotIn()` — аналогичен методу `orWhereIn()`;
- `orWherePivotNotIn()` — аналогичен методу `orWhereNotIn()`;
- `wherePivotNull()` — аналогичен методу `whereNull()`;
- `wherePivotNotNull()` — аналогичен методу `whereNotNull()`;
- `orWherePivotNull()` — аналогичен методу `orWhereNull()`;
- `orWherePivotNotNull()` — аналогичен методу `orWhereNotNull()`.

Объект связи «многие-со-многими» также поддерживает метод `orderByPivot()`, аналогичный методу `orderBy()` (см. разд. 7.3.5) и сортирующий связанные записи по значению указанного поля связующей таблицы:

```
public function sparesMoreThenOne() {
    return $this->belongsToMany(Spare::class)
                . . .
                ->orderByPivot('cnt', 'desc');
}
```

16.4. Полиморфные связи

Обычная межтабличная связь устанавливается между строго определенными таблицами (например, `rubrics` и `bbs`). Напротив, *полиморфная* (или *обобщенная*) связь позволяет связать запись таблицы, в которой она объявлена, с записью любой из таблиц, что имеются в базе данных, при этом первая таблица станет вторичной, а вторая — первичной.

16.4.1. Создание поля внешнего ключа для полиморфной связи

Для создания поля внешнего ключа, участвующего в установлении полиморфной связи, применяется один из приведенных далее методов, поддерживаемых классом структуры таблицы и вызываемых в коде миграции:

- `morphs(<имя поля>[, <имя индекса>=null])` — создает обычное полиморфное поле внешнего ключа с заданным *именем* и индекс, основанный на этом поле. Если *имя индекса* не указано, оно будет сгенерировано самим фреймворком;
- `nullableMorphs()` — то же самое, что и `morphs()`, только создает поле, необязательное для заполнения;
- `uuidMorphs()` — то же самое, что и `morphs()`, только создает поле, предназначенное для хранения ключей в виде универсальных уникальных идентификаторов;
- `nullableUuidMorphs()` — то же самое, что и `uuidMorphs()`, только создает поле, необязательное для заполнения.

Не забываем, что поле внешнего ключа создается во вторичной таблице.

Каждый из этих методов фактически создает в таблице два поля:

- *<имя создаваемого поля>*`_id` — целочисленное беззнаковое размером 8 байтов — для хранения непосредственно ключа связываемой записи;
- *<имя создаваемого поля>*`_type` — строковое, длиной 255 символов — для хранения типа записи (по умолчанию в качестве типа используется путь к классу модели, к которой относится связываемая запись).

Пример кода миграции, создающего полиморфное поле внешнего ключа:

```
Schema::create('tags', function (Blueprint $table) {
    . . .
    $table->morphs('taggable');
});
```

Поля, участвующие в установлении полиморфной связи, можно создать и полностью вручную, дав им произвольные имена:

```
Schema::create('tags', function (Blueprint $table) {
    . . .
    $table->string('tag_model');
    $table->unsignedBigInteger('tag_key');
    $table->index(['tag_model', 'tag_key']);
});
```

Для удаления полиморфного поля внешнего ключа служит метод `dropMorphs()`, имеющий тот же формат вызова, что и метод `morphs()`:

```
Schema::table('tags', function (Blueprint $table) {
    $table->dropMorphs('taggable');
});
```

16.4.2. Создание полиморфных связей

Полиморфные связи создаются так же, как и обычные, — объявлением в классах связываемых моделей особых методов, устанавливающих связь и являющихся общедоступными.

16.4.2.1. Полиморфная связь «один-со-многими»

1. В каждом классе первичной модели — объявляется метод, устанавливающий «прямую» связь со вторичной моделью. Он должен возвращать объект «прямой» связи, выдаваемый методом `morphMany()` модели:

   ```
   morphMany(<имя класса связываемой вторичной модели>,
             <имя полиморфного поля внешнего ключа>[,
             <имя поля вторичной таблицы, хранящего тип записи>=null[,
             <имя поля вторичной таблицы, хранящего ключ>=null[,
             <имя ключевого поля первичной модели>=null]]])
   ```

   Если во вторичной таблице:

   - было создано полиморфное поле внешнего ключа (вызовом метода `morph()` или аналогичного) — следует указать *имя* этого *полиморфного поля*. Тогда *имена полей вторичной таблицы*, хранящих *тип связываемой записи* и ее *ключ*, задавать не нужно. Фреймворк предположит, что эти поля имеют имена форматов *<имя полиморфного поля>*`_type` и *<имя полиморфного поля>*`_id`;
   - были созданы отдельные поля для хранения типа и ключа связываемой записи — следует указать *имена полей вторичной таблицы*, хранящих *тип связываемой записи*, и ее *ключ*. *Имя полиморфного поля внешнего ключа* может быть произвольным (например, «пустой» строкой).

   Если не указано *имя ключевого поля*, будет взято ключевое поле, заданное в классе текущей модели.

2. В классе вторичной модели — объявляется метод, создающий «обратную» связь со всеми первичными моделями. Он должен возвращать объект «обратной» связи, выданный методом `morphTo()` модели:

   ```
   morphTo([<имя полиморфного поля внешнего ключа>=null[,
            <имя поля вторичной таблицы, хранящего тип записи>=null[,
            <имя поля вторичной таблицы, хранящего ключ>=null[,
            <имя ключевого поля первичных моделей>=null]]]])
   ```

   Аналогично, если во вторичной таблице:

   - было создано полиморфное поле внешнего ключа — следует указать *имя* этого *полиморфного поля*. *Имена полей вторичной таблицы*, хранящих *тип* и *ключ* связанной записи, задавать не нужно.

Имя полиморфного поля внешнего ключа можно и не указывать — тогда Laravel использует в его качестве имя метода, создающего обратную связь;

- были созданы отдельные поля для хранения типа и ключа связываемой записи — следует указать *имена полей вторичной таблицы*, хранящих *тип* и *ключ*. *Имя полиморфного поля внешнего ключа* может быть произвольным.

Имя ключевого поля указывается сразу для всех первичных моделей, и если оно не задано, будут использоваться ключевые поля, заданные в классах первичных моделей.

Пример установления полиморфной связи «один-со-многими» между первичными моделями `Rubric` и `Bb` и вторичной `Tag` (хранит теги) при условии, что во вторичной таблице было создано полиморфное поле внешнего ключа `taggable`:

```
use App\Models\Tag;
class Rubric extends Model {
    . . .
    public function tags() {
        return $this->morphMany(Tag::class, 'taggable');
    }
}

use App\Models\Tag;
class Bb extends Model {
    . . .
    public function tags() {
        return $this->morphMany(Tag::class, 'taggable');
    }
}

class Tag extends Model {
    . . .
    public function taggable() {
        return $this->morphTo();
    }
}
```

Пример установления аналогичной связи в случае, если были созданы отдельные поля: `tag_model` и `tag_key`:

```
use App\Models\Tag;
class Rubric extends Model {
    . . .
    public function tags() {
        return $this->morphMany(Tag::class, '', 'tag_model', 'tag_key');
    }
}

use App\Models\Tag;
class Bb extends Model {
    . . .
```

```
    public function tags() {
        return $this->morphMany(Tag::class, '', 'tag_model', 'tag_key');
    }
}

class Tag extends Model {
    . . .
    public function taggable() {
        return $this->morphTo('', 'tag_model', 'tag_key');
    }
}
```

16.4.2.2. Полиморфная связь «один-с-одним из многих»

В классе первичной модели объявляется метод, формирующий «прямую» связь со вторичной моделью. Он должен создавать объект «прямой» связи с помощью метода `morphOne()` модели, чей формат вызова аналогичен таковому у метода `morphMany()` (см. *разд. 16.4.2.1*). У созданного таким образом объекта связи следует вызвать один из методов, описанных в *разд. 5.4.2*.

Пример установления полиморфной связи «один-с-одним из многих» между первичной моделью `Bb` и вторичной `Tag`, выдающей наиболее «свежий» из добавленных тегов:

```
class Bb extends Model {
    . . .
    public function tags() {
        return $this->morphOne(Tag::class, 'taggable')->latestOfMany();
    }
}
```

16.4.2.3. Полиморфная связь «один-с-одним»

1. В классе первичной модели — объявляется метод, формирующий «прямую» связь со вторичной моделью. Он должен возвращать объект «прямой» связи, выданный методом `morphOne()` (описан в *разд. 16.4.2.2*).
2. В классе вторичной модели — объявляется такой же метод, формирующий «обратную» связь, что и во вторичной модели, связанной связью «один-со-многими» (см. *разд. 16.4.2.1*).

Пример создания полиморфной связи «один-с-одним» между стандартной первичной моделью `User` и вторичной моделью `Thumbnail` (графическая миниатюра):

```
Schema::create('thumbnails', function (Blueprint $table) {
    . . .
    $table->morphs('thumbnailable');
    . . .
});

use App\Models\Thumbnail;
class User extends Model {
    public function thumbnail() {
```

```
        return $this->morphOne(Thumbnail::class, 'thumbnailable');
    }
}

class Thumbnail extends Model {
    public function thumbnailable() {
        return $this->morphTo();
    }
}
```

16.4.2.4. Полиморфная связь «многие-со-многими»

Полиморфная связь «многие-со-многими» позволяет связать произвольное количество записей одной таблицы (назовем ее *ведущей*) с произвольным количеством записей любой из остальных таблиц (*ведомых*).

1. Объявляется связующая таблица — содержащая два поля:
   - обычное поле внешнего ключа — для хранения ключа связанной записи ведущей таблицы;
   - полиморфное поле внешнего ключа — для хранения типа и ключа связанной записи ведомых таблиц.
2. В каждом классе ведомой модели — объявляется метод, формирующий связь с ведущей моделью посредством связующей таблицы. Он должен возвращать результат вызова метода `morphToMany()` модели:

```
morphToMany(<имя класса связываемой ведущей модели>,
            <имя полиморфного поля внешнего ключа связующей таблицы>[,
            <имя связующей таблицы>=null[,
            <имя поля связующей таблицы, хранящего ключ записи ↵
             ведомой таблицы>=null[,
            <имя поля связующей таблицы, хранящего ключ записи ↵
             ведущей таблицы>=null[,
            <имя ключевого поля ведомой таблицы>=null[,
            <имя ключевого поля ведущей таблицы>=null]]]]])
```

   Если в связующей таблице:

   - было создано полиморфное поле внешнего ключа для связи с ведомыми таблицами — следует указать *имя* этого *полиморфного поля*. Тогда фреймворк предположит, что *поля связующей таблицы*, хранящие *тип* и *ключ* связываемой записи, имеют имена форматов *<имя полиморфного поля>*`_type` и *<имя полиморфного поля>*`_id`;
   - были созданы отдельные поля для хранения типа и ключа связываемой записи ведомой таблицы — следует указать *имена полей связующей таблицы*, хранящих *тип связываемой записи* и ее *ключ*. *Имя полиморфного поля внешнего ключа* может быть произвольным (например, «пустой» строкой).

   Если не указано *имя связующей таблицы*, Laravel предполагает, что ее имя совпадает с *именем полиморфного поля*, приведенным к множественному числу (например, если указано имя полиморфного поля `taggable`, предполагается, что связующая таблица

называется `taggables`). Если не указаны *имена ключевых полей*, будут взяты имена, записанные в классах соответствующих моделей.

3. В классе ведущей модели — объявляются методы, каждый из которых формирует связь с одной из ведомых моделей. Такой метод должен возвращать результат вызова метода `morphedByMany()` модели:

```
morphedByMany(<имя класса связываемой ведомой модели>,
              <имя полиморфного поля внешнего ключа связующей ↵
               таблицы>[,
              <имя связующей таблицы>=null[,
              <имя поля связующей таблицы, хранящего ключ записи ↵
               ведомой таблицы>=null[,
              <имя поля связующей таблицы, хранящего ключ записи ↵
               ведущей таблицы>=null[,
              <имя ключевого поля ведомой таблицы>=null[,
              <имя ключевого поля ведущей таблицы>=null]]]]])
```

Параметры здесь те же, что и у метода `morphMany()`.

Пример установления полиморфной связи «один-со-многими» между ведомыми моделями `Rubric` и `Bb` и ведущей `Tag2` (хранит уникальные теги, каждый из которых может быть связан с произвольным количеством записей):

```
Schema::create('tag2s', function (Blueprint $table) {
    $table->id();
    $table->string('name', 20);
    $table->timestamps();
});

Schema::create('taggables', function (Blueprint $table) {
    $table->foreignId('tag2_id')->constrained()->cascadeOnDelete();
    $table->morphs('taggable');
});

use App\Models\Tag2;
class Rubric extends Model {
    . . .
    public function tags2() {
        return $this->morphToMany(Tag2::class, 'taggable');
    }
}

// В модели Bb правки будут аналогичными правкам в модели Rubric

use App\Models\Rubric;
use App\Models\Bb;
class Tag2 extends Model {
    public function rubrics() {
        return $this->morphedByMany(Rubric::class, 'taggable');
    }
```

```
    public function bbs() {
        return $this->morphedByMany(Bb::class, 'taggable');
    }
}
```

16.4.3. Работа с записями, связанными полиморфной связью

Для работы с записями, связанными полиморфной связью, применяются те же инструменты, что используются для обращения с записями, связанными обычной связью (см. *разд. 6.1.4* и *7.4*). Вот несколько примеров:

```
>>> use App\Models\Bb;
>>> // Добавим пару тегов к первому объявлению, применив модель Tag
>>> $bb = Bb::first();
>>> echo $bb->title;
Гараж
>>> use App\Models\Tag;
>>> // Создаем и добавляем первый тег
>>> $tag = new Tag(['tag' => 'дом']);
>>> $tag->taggable()->associate($bb);
>>> $tag->save();
>>> // Создаем и добавляем второй тег
>>> $bb->tags()->save(new Tag(['tag' => 'жилище']));
>>> // Посмотрим, какие теги привязаны к первому объявлению
>>> foreach ($bb->tags()->get() as $tag) echo $tag->tag, ' | ';
дом | жилище |
>>> // Добавим тег "машина" к объявлению о продаже "Запорожца"
>>> $bb = Bb::firstWhere('title', 'Запорожец');
>>> $bb->tags()->create(['tag' => 'машина']);
>>> // Добавим тег "техника" к одноименной рубрике
>>> $rubric = Rubric::firstWhere('name', 'Техника');
>>> $rubric->tags()->create(['tag' => 'техника']);

>>> // Создадим тег "машина", применив модель Tag2, и привяжем его к двум
>>> // объявлениям…
>>> use App\Models\Tag2;
>>> $tag2 = new Tag2(['name' => 'машина']);
>>> $tag2->save();
>>> $bb = Bb::firstWhere('title', 'ГАЗ');
>>> $bb->tags2()->attach($tag2->id);
>>> $bb = Bb::firstWhere('title', 'Запорожец');
>>> $bb->tags2()->attach($tag2->id);
>>> // …и одной рубрике
>>> use App\Models\Rubric;
>>> $rubric = Rubric::firstWhere('name', 'Транспорт');
>>> $rubric->tags2()->attach($tag2->id);
>>> // Посмотрим, к каким объявлениям и рубрикам привязан этот тег
>>> $tag2->bbs()->pluck('title');
=> [ "Запорожец", "ГАЗ" ]
```

```
>>> $tag2->rubrics()->pluck('name');
=> [ "Транспорт" ]
```

Поддерживается фильтрация записей вторичной модели по наличию или отсутствию связанных с ними записей первичных моделей. Для этого используются следующие методы, аналогичные описанным в *разд. 7.4*:

- `whereHasMorph()` — аналогичен методу `whereHas()`:

  ```
  whereHasMorph(<имя связи>, <массив с путями к классам моделей>[,
                <анонимная функция, отбирающая связанные записи>=null])
  ```

  В расчет будут приниматься только записи из тех моделей, пути к классам которых приведены в заданном *массиве*. Если вместо *массива* задать строку `'*'`, в расчет будут приниматься записи любых моделей. *Анонимная функция* должна принимать первым параметром объект построителя запросов, а вторым — строку с путем к классу модели. Примеры:

  ```
  >>> $tags = Tag::whereHasMorph('taggable', '*')->pluck('tag');
  => [ "дом", "жилище", "машина", "техника" ]
  >>> $tags = Tag::whereHasMorph('taggable', [App\Models\Rubric::class])
  ...             ->pluck('tag');
  => [ "техника" ]
  >>> $tags = Tag::whereHasMorph('taggable', '*',
  ...     function ($query, $type) {
  ...         if ($type == App\Models\Bb::class)
  ...             $query->where('price', '>', 100000);
  ... })->pluck('tag');
  => [ "дом", "жилище", "техника" ]
  ```

- `orWhereHasMorph()` — аналогичен методу `orWhereHas()`. Формат вызова такой же, как и у метода `whereHasMorph()`;

- `whereDoesntHaveMorph()` — аналог метода `whereDoesntHave()`. Формат вызова такой же, как и у метода `whereHasMorph()`:

  ```
  >>> $tags = Tag::whereDoesntHaveMorph('taggable',
  ...                                   [App\Models\Bb::class],
  ...     function ($query, $type) {
  ...         $query->where('price', '>', 100000);
  ... })->pluck('tag');;
  => [ "машина" ]
  ```

- `orWhereDoesntHaveMorph()` — аналог метода `orWhereDoesntHave()`. Формат вызова такой же, как и у метода `whereHasMorph()`.

16.4.4. Указание своих типов связываемых записей

По умолчанию в качестве типов записей, записываемых в полиморфные поля внешнего ключа, используются строковые пути к классам моделей, к которым относятся связываемые записи. Так, если запись относится к модели `Rubric`, в соответствующее полиморфное поле в качестве типа будет записана строка `'App\\Models\\Rubric'`.

Можно указать фреймворку использовать другие обозначения типов записей. Это может понадобиться, например, если поле типа записи имеет ограниченный размер или вообще не является строковым, а также чтобы не привязывать типы записей к структуре проекта.

Типы записей указываются в теле метода `boot()` провайдера `App\Providers\AppServiceProvider` (или любого другого провайдера, в том числе созданного вручную) посредством вызова у класса `Illuminate\Database\Eloquent\Relations\Relation` статического метода `enforceMorphMap()`:

```
enforceMorphMap(<ассоциативный массив типов>[,
                <объединять массивы?>=true])
```

> ***Примечание***
>
> Метод `morphMap()`, использовавшийся для этого в предыдущих версиях фреймворка, в Laravel 9 применять не рекомендуется.

В *ассоциативном массиве типов* ключи элементов должны соответствовать задаваемым типам записей, а значения — представлять собой пути к классам соответствующих моделей. Если параметру *объединять массивы* дать значение `false`, указанный в вызове метода *массив* заменит массив, заданный в предыдущих вызовах этого метода (по умолчанию массивы объединяются). Пример:

```
use Illuminate\Database\Eloquent\Relations\Relation;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Relation::enforceMorphMap([
            'rubric' => App\Models\Rubric::class,
            'bb' => App\Models\Bb::class
        ]);
    }
}
```

> ***Сохраненные ранее в таблицах типы записей не изменятся...***
>
> ...после указания своих типов. Их придется менять вручную путем правки записей.

Выяснить тип записей, назначенный той или иной модели, можно, вызвав у ее объекта метод `getMorphClass()`:

```
$rubricRecordType = $rubric->getMorphClass();
$bbRecordType = $bb->getMorphClass();
```

Узнать полный путь к классу модели по назначенному ей *типу* можно, использовав статический метод `getMorphedModel(<тип записей>)` класса `Relation`:

```
use Illuminate\Database\Eloquent\Relations\Relation;
$rubricModelPath = Relation::getMorphedModel('rubric');
```

16.5. Пределы

Пределом (scope) в Laravel называются заранее созданные условия фильтрации и сортировки записей, которые накладываются на выбираемые записи автоматически или по требованию программиста. Пределы пригодятся, если в разных местах кода требуется выбирать записи, удовлетворяющие одним и тем же условиям.

16.5.1. Локальные пределы

Локальный предел накладывает заданные в нем условия фильтрации и сортировки только по требованию программиста. Он должен представлять собой общедоступный метод модели, который:

- имеет имя, начинающееся с префикса `scope`;
- принимает по крайней мере один параметр — объект построителя запросов, с помощью которого задаются условия фильтрации и сортировки записей;
- возвращает объект построителя запроса, полученный с первым параметром, или ничего не возвращает;
- может получать с остальными параметрами другие необходимые для работы значения.

Пример объявления в модели `Bb` двух локальных пределов — `inRubric()` и `expensive()`:

```
class Bb extends Model {
    . . .
    public function scopeInRubric($query, $rubric) {
        return $query->where('rubric_id', $rubric->id);
    }

    public function scopeExpensive($query) {
        $query->where('price', '>', 1000000);
    }
}
```

Применить локальный предел можно, вызвав у модели или у построителя запросов метод с именем, совпадающим с именем самого предела, только начинающимся со строчной буквы и без префикса `scope`:

```
>>> use App\Models\Bb;
>>> $bbs = Bb::expensive()->pluck('price');
=> [ "5000000.0", "4000000.0", "70000000.0", "2000000.0" ]
```

Дополнительные параметры, принимаемые локальным пределом, указываются в вызове этого метода:

```
>>> use App\Models\Rubric;
>>> $rubric = Rubric::firstWhere('name', 'Легковой');
>>> $bbs = Bb::inRubric($rubric)->pluck('title');
=> [ "Жигули", "Запорожец" ]
```

При последовательном вызове пределов они объединяются с помощью логического оператора И:

```
>>> $bbs = Bb::inRubric($rubric)->expensive()->get();
>>> foreach ($bbs as $bb) echo $bb->title, ': ', $bb->price . "\r\n";
>>> // Ничего выведено не будет
```

Чтобы объединить их с помощью оператора ИЛИ, следует использовать метод `orWhere()` с анонимной функцией:

```
>>> $bbs = Bb::inRubric($rubric)
...          ->orWhere(function ($query) { $query->expensive(); })
...          ->get();
. . .
Дом: 5000000
Жигули: 1000000
Запорожец: 10000
МАЗ: 4000000
ГАЗ: 70000000
```

Вместо этой громоздкой конструкции можно применить «связку» `orWhere`:

```
>>> $bbs = Bb::inRubric($rubric)->orWhere->expensive()->get();
```

16.5.2. Глобальные пределы

Глобальный предел накладывает заданные в нем условия автоматически на все запросы, выполненные с участием модели, с которой он связан.

Глобальный предел может быть реализован двумя способами:

- в виде класса.

  Класс глобального предела с указанным *именем* создается командой:

  ```
  php artisan make:scope <имя класса>
  ```

  По соглашениям имя класса глобального предела должно заканчиваться суффиксом `Scope`. Впрочем, это необязательно и на работу проекта никак не влияет.

  Класс глобального предела объявляется в пространстве имен `App\Models\Scopes` (соответствующая папка создается автоматически). Также его можно объявить в пространстве имен `App\Scopes` (как было в предыдущих версиях Laravel).

  Класс глобального предела должен реализовывать интерфейс `Illuminate\Database\Eloquent\Scope`. В классе должен присутствовать общедоступный метод `apply()`, в качестве параметров принимающий объект построителя запросов и «пустой» объект модели, задающий необходимые условия фильтрации и сортировки с помощью полученного построителя запросов и не возвращающий результата.

  В листинге 16.1 показан код глобального предела `App\Models\Scopes\BbsScope`, отбирающего только объявления, которые помечены как подлежащие публикации, и сортирующего их по убыванию даты создания.

Листинг 16.1. Пример глобального предела

```
namespace App\Models\Scopes;
use Illuminate\Database\Eloquent\Builder;
```

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;
class BbsScope implements Scope {
    public function apply(Builder $builder, Model $model) {
        $builder->where('publish', true)->latest();
    }
}
```

Созданный таким образом глобальный предел следует зарегистрировать в модели. Это выполняется в защищенном, статическом, не принимающем параметров методе `booted()` модели, вызовом статического же метода `addGlobalScope(<`*`объект глобального предела`*`>)`. Конструктор глобального предела вызывается без параметров. Пример:

```
use App\Models\Scopes\BbsScope;
class Bb extends Model {
    . . .
    protected static function booted() {
        static::addGlobalScope(new BbsScope);
    }
}
```

- в виде анонимной функции — непосредственно в классе модели. Как правило, в таком виде создаются простые глобальные пределы.

  Глобальный предел, оформленный в виде анонимной функции, также регистрируется в теле защищенного статического метода `booted()` модели вызовом того же статического метода `addGlobalScope()`, но в другом формате:

  `addGlobalScope(<`*`обозначение`*`>, <`*`анонимная функция`*`>)`

  Произвольное *`обозначение`* глобального предела задается в виде строки и должно быть уникальным. *`Анонимная функция`*, реализующая предел, должна принимать в качестве параметра объект построителя запросов. Пример:

```
use Illuminate\Database\Eloquent\Builder;
class Bb extends Model {
    . . .
    protected static function booted() {
        static::addGlobalScope('bbs', function (Builder $builder) {
            $builder->where('publish', true)->latest();
        });
    }
}
```

В каком бы виде ни был реализован глобальный предел, он начинает действовать сразу после регистрации. Таким образом, набрав код:

```
>>> use App\Models\Bb;
>>> $bbs = Bb::all();
```

мы получим в переменной `bbs` коллекцию записей, хранящих только опубликованные объявления и отсортированных по убыванию даты создания.

Чтобы временно отключить какой-либо глобальный предел, следует вызвать один из двух приведенных далее методов, поддерживаемых построителем запросов и возвращающих в качестве результата его текущий объект:

- `withoutGlobalScope(<обозначение>)` — отключает глобальный предел с заданным *обозначением*, в качестве которого можно указать как собственно обозначение предела, реализованного анонимной функцией, так и полный путь к классу предела:

```
$bbs = Bb::withoutGlobalScope(App\Models\Scopes\BbsScope::class)
         ->all();
```

- `withoutGlobalScopes([<массив обозначений>=null])` — отключает глобальные пределы с присутствующими в *массиве* обозначениями. Если *массив* не указан, отключает все глобальные пределы, зарегистрированные в модели. Пример:

```
$bbs = Bb::withoutGlobalScopes(['bbs',
                        App\Models\Scopes\SpecialScope::class])
         ->all();
```

16.6. Выполнение «сырых» SQL-запросов

«Сырым» называется SQL-запрос, не сгенерированный построителем запросов, а набранный самим программистом в виде SQL-кода. В формате «сырых» запросов оформляются вызовы специфических команд и функций, поддерживаемых определенными СУБД, а также особо сложные или специальные запросы, которые построитель сгенерировать не может.

16.6.1. «Сырые» вызовы функций СУБД

Вызов какой-либо функции СУБД можно оформить в виде описанного в *разд. 7.5.2* метода `raw(<SQL-код вызова функции>)` фасада `Illuminate\Support\Facades\DB`. Этот вызов можно использовать практически в любом методе построителя запросов: `select()`, `where()`, `orderBy()` и др. Пример:

```
>>> use Illuminate\Support\Facades\DB;
>>> $bbs = Bb::select(DB::raw('printf("%10.0f руб.", price) as p'))
...          ->get();
>>> foreach ($bbs as $bb) { echo $bb->p, "\r\n"; }
   1000000 руб.
   5000000 руб.
. . .
```

16.6.2. «Сырые» команды SQL

Для вставки в SQL-запрос в «сыром» виде отдельных команд: `SELECT`, `WHERE`, `ORDER BY` — служат следующие методы, поддерживаемые построителем запросов:

- `selectRaw(<SQL-код>[, <массив параметров>=null])` — создает «сырую» команду выборки `SELECT` на основе заданного *SQL-кода*.

  Если в *SQL-код* необходимо подставить параметры, чьи значения вычисляются в процессе работы сайта, в нужных местах кода следует вставить литералы `?` (вопро-

сительный знак — обозначает позиционный параметр) или :*<имя>* (обозначает именованный параметр). Значения этих параметров приводятся в задаваемом *массиве*: индексированном, если параметры позиционные, или ассоциативном, если параметры именованные. Примеры:

```
>>> echo Bb::selectRaw('count(*) as cnt')->get()[0]['cnt'];
11

>>> // Используем позиционный параметр, код компактнее
>>> $bbs = Bb::selectRaw('price, price * ? as discounted', [0.95])
...          ->get();
>>> foreach ($bbs as $bb)
...     echo $bb->price, ' -> ', $bb->discounted, "\r\n";
1000000 -> 950000
5000000 -> 4750000
. . .

>>> // Используем именованный параметр, код нагляднее
>>> $bbs = Bb::selectRaw('price, price * :discount as discounted',
...                      ['discount' => 0.95])->get();
```

- `whereRaw()` — создает «сырую» команду фильтрации `WHERE` на основе заданных *SQL-кода* и *массива параметров*:

```
whereRaw(<SQL-код>[, <массив параметров>=null[,
                     <логический оператор>='and']])
```

Команда, созданная текущим вызовом этого метода, объединяется с командой, созданной его предыдущим вызовом, с применением заданного *логического оператора* (по умолчанию: `AND`). Примеры:

```
>>> $bbs = Bb::whereRaw('(price >= ? AND price <= ?)',
...                     [1000000, 5000000])
...          ->whereRaw('(rubric_id = ?)', [2])->get();
=> [ App\Models\Bb { ... price: "5000000.0", rubric_id: "2"} ]
>>> $bbs = Bb::whereRaw('(price >= ? AND price <= ?)',
...                      [1000000, 5000000])
...          ->whereRaw('(rubric_id = ?)', [2], 'or')->get();
=> [ App\Models\Bb { ... price: "1000000.0", rubric_id: "7" },
...
     App\Models\Bb { ... price: "200000.0", rubric_id: "2" },
...
```

- `orWhereRaw()` — то же самое, что и `whereRaw()`, только создаваемая им команда объединяется с предыдущей с использованием логического оператора `OR`. Формат вызова такой же, как и у метода `selectRaw()`;
- `orderByRaw()` — создает «сырую» команду сортировки `ORDER BY` на основе заданных *SQL-кода* и *массива параметров*. Формат вызова такой же, как и у метода `selectRaw()`. Пример:

```
>>> $rubrics = Rubric::orderByRaw('length(name)')->get();
>>> foreach ($rubrics as $rubric) echo $rubric->name, ' | ';
Дома | Дачи | Здания | Гаражи | Техника | Бытовая | Легковой | ↳
Грузовой | Транспорт | Служебные |
```

- `groupByRaw()` — создает «сырую» команду группировки `GROUP BY` на основе заданных *SQL-кода* и *массива параметров*. Формат вызова такой же, как и у метода `selectRaw()`;
- `havingRaw()` — создает «сырую» команду фильтрации групп `HAVING` на основе заданных *SQL-кода* и *массива параметров*, объединяя ее с предыдущей командой с помощью указанного *логического оператора* (по умолчанию: `AND`). Формат вызова такой же, как и у метода `whereRaw()`.
- `orHavingRaw()` — то же самое, что и `havingRaw()`, только создаваемая им команда объединяется с предыдущей с использованием логического оператора `OR`. Формат вызова такой же, как и у метода `selectRaw()`.

  Пример:

```
>>> $result = Bb::selectRaw('rubric_id, count(*) as cnt')
...             ->groupByRaw('rubric_id')->havingRaw('count(*) >= 2')
...             ->get();
>>> foreach ($result as $r)
...     echo $r->rubric->name, ': ', $r->cnt, "\r\n";
Грузовой: 3
Легковой: 2
Дома: 3
```

16.5.3. «Сырые» SQL-запросы целиком

Для выполнения целых «сырых» SQL-запросов применяются следующие методы фасада `DB`:

- `select(<SQL-код>[, <массив параметров>=null])` — создает «сырой» SQL-запрос на выборку записей, основываясь на заданных *SQL-коде* и *массиве параметров*. В качестве результата возвращает массив объектов встроенного в PHP класса `stdClass`, каждый из которых хранит одну выбранную запись. Пример:

```
>>> $bbs = DB::select('SELECT title, content, price FROM bbs ' .
...             'WHERE price > ? ORDER BY created_at DESC', [2000000]);
=> [ {#4749 +"title": "Дом", +"content": "Большой, трехэтажный",
            +"price": 5000000.0},
     {#4753 +"title": "ГАЗ", +"content": "Совсем новый",
            +"price": 70000000.0},
     {#4734 +"title": "МАЗ", +"content": "Старый, заслуженный",
            +"price": 4000000.0} ]
```

- `scalar()` — создает «сырой» запрос на извлечение единственного значения, которое и возвращает. Формат вызова такой же, как и у метода `select()`. Пример:

```
>>> $cnt = DB::scalar('SELECT COUNT(*) FROM bbs WHERE publish=false');
=> 2
```

- `insert()` — создает «сырой» запрос на добавление записей. Формат вызова такой же, как и у метода `select()`. В качестве результата возвращает `true`, если записи были успешно добавлены, и `false` — в противном случае. Пример:

```
>>> DB::insert('INSERT INTO rubrics (name) VALUES (?)', ['Игрушки']);
```

- `update()` — создает «сырой» запрос на правку записей. Формат вызова такой же, как и у метода `select()`. В качестве результата возвращает количество исправленных записей. Пример:

```
>>> DB::update('UPDATE bbs SET price = ? WHERE id = ?', [20000, 4]);
```

- `delete()` — создает «сырой» запрос на удаление записей. Формат вызова такой же, как и у метода `select()`. В качестве результата возвращает количество удаленных записей. Пример:

```
>>> DB::delete('DELETE FROM bbs WHERE id = ?', [15]);
```

- `statement()` — создает «сырой» запрос на выполнение операции, отличной от выборки, добавления, правки и удаления записей. Формат вызова такой же, как и у метода `select()`. В качестве результата возвращает `true`, если операция была успешно выполнена, и `false` — в противном случае. Пример:

```
>>> DB::statement('DROP TABLE temporary_table');
```

- `unprepared(<SQL-код>)` — создает «сырой» запрос, основываясь на заданном *SQL-коде*. Заданный *код* не подвергается никакой предварительной обработке, в частности, в нем не выполняется подстановка параметров. Пример:

```
DB::unprepared('INSERT INTO rubrics (name) VALUES ("Игрушки")');
```

16.7. Блокировка записей

Иногда бывает необходимо в процессе выполнения запроса заблокировать выбираемые записи на правку или даже чтение другими запросами. Для этого построитель запросов поддерживает пару методов, возвращающих его текущий объект:

- `sharedLock()` — накладывает на записи *разделяемую блокировку*, не позволяющую править записи до окончания выполнения текущего запроса. Тем не менее позволяет другим запросам читать записи и накладывать на них разделяемые блокировки. Пример:

```
>>> $bbs = Bb::where('price', '>', 100000)->sharedLock()->get();
```

- `lockForUpdate()` — накладывает на записи *исключительную блокировку*, не позволяющую ни править, ни читать записи, ни накладывать на них блокировку до окончания выполнения текущего запроса.

Как только запрос, блокирующий записи, выполнится, блокировка будет автоматически снята.

16.8. Управление транзакциями

Если в процессе выполнения клиентского запроса необходимо добавить, исправить и (или) удалить сразу несколько записей, необходимые операции удобнее заключить в *транзакцию*. Это гарантирует, что либо все операции будут выполнены, либо, в случае возникновения ошибки, не будет выполнена ни одна, и целостность базы данных не будет нарушена.

Laravel предоставляет две разновидности инструментов для управления транзакциями.

16.8.1. Автоматическое управление транзакциями

Автоматическое управление транзакциями, применимое в большинстве случаев, реализует метод `transaction()` фасада `DB`:

```
transaction(<анонимная функция>[, <количество попыток завершения>=1])
```

В теле заданной *анонимной функции*, не принимающей параметров, записываются операции, которые должны быть заключены в транзакцию. В начале выполнения тела этой функции транзакция автоматически запускается, а в конце — подтверждается или, в случае возникновения ошибки, откатывается.

Если возможно возникновение взаимоблокировок (deadlock), можно указать количество *попыток завершения транзакции* (по умолчанию: всего одна). Транзакция будет подвергнута откату, если все эти попытки не увенчаются успехом. Пример:

```
use Illuminate\Support\Facades\DB;
DB::transaction(function () {
    Bb::where('created_at', '<', '2020-12-31')
      ->update(['publish' => false]);
    Bb::where('created_at', '<', '2017-12-31')->delete();
});
```

16.8.2. Ручное управление транзакциями

Для более специфических решений, возможно, придется управлять транзакциями вручную, вызывая следующие методы фасада `DB`:

- `beginTransaction()` — запускает транзакцию;
- `commit()` — подтверждает транзакцию;
- `rollBack()` — откатывает транзакцию;
- `transactionLevel()` — возвращает количество активных транзакций.

Пример:

```
DB::beginTransaction();
try {
    Bb::where('created_at', '<', '2020-12-31')
      ->update(['publish' => false]);
    Bb::where('created_at', '<', '2017-12-31')->delete();
    DB::commit();
} catch {$e} {
    DB::rollBack();
}
```

16.9. Очистка моделей

Очистка моделей (pruning) — это автоматическое, обычно выполняемое по расписанию удаление записей, отвечающих заданным условиям фильтрации (например, объявлений, оставленных десять лет назад и соответственно неактуальных).

Очистка моделей реализуется непосредственно в их классах и бывает двух видов:

- *обычная* — выполняемая средствами моделей. Производя такую очистку, Laravel извлекает объекты моделей, представляющие подлежащие удалению записи, и вызывает у каждого из них метод `delete()`. При этом генерируются события модели, кроме того, имеется возможность перед собственно очисткой выполнить какие-либо предварительные действия (например, удалить файлы, сохраненные в удаляемых записях). Однако обычная очистка выполняется весьма медленно и при удалении множества записей отнимает много времени.

  Чтобы реализовать в классе модели обычную очистку, следует присоединить к этому классу трейт `Illuminate\Database\Eloquent\Prunable`;

- *массовая* — выполняемая средствами СУБД в обход моделей. Производится очень быстро. Однако при этом метод `delete()` модели не вызывается (соответственно реализованная в нем дополнительная функциональность, если такая есть, не будет задействована), события модели не генерируются, и никакие предварительные действия перед очисткой также выполнить нельзя.

  Для реализации в модели массовой очистки следует присоединить к ее классу трейт `Illuminate\Database\Eloquent\MassPrunable`.

Программный код, выполняющий очистку, записывается в следующих методах класса модели:

- `prunable()` — общедоступный, не должен принимать параметров. Должен возвращать объект построителя запросов с указанными в нем условиями фильтрации, которым должны удовлетворять записи, подлежащие удалению в процессе очистки;
- `schedule(Schedule $schedule)` — защищенный, должен принимать с параметром `schedule` объект класса `Illuminate\Console\Scheduling\Schedule`, представляющий планировщик заданий Laravel (будет описан в *главе 27*). Должен указывать этому объекту запускать команду утилиты artisan `model:prune` и задавать расписание ее запуска. Не должен возвращать результат;
- `pruning()` — защищенный, не должен принимать параметров и возвращать результат. Должен выполнять предварительные действия перед собственно очисткой. Если же таких действий производить не нужно, объявлять этот метод необязательно.

  В моделях, реализующих массовую очистку, этот метод не вызывается.

Удаление записей выполняется отдельными чанками. Размер такого чанка по умолчанию: 1000 записей (может быть изменен в команде `model:prune`). Можно указать другой размер, задав его в общедоступном свойстве `prunableChunkSize` класса модели.

Пример кода, вставленного в класс модели `Bb`, выполняющего обычную очистку каждый год, удаляющего объявления, которые были оставлены десять лет назад, и задающего размер чанка 500 записей:

```
use Illuminate\Database\Eloquent\Prunable;
use Illuminate\Console\Scheduling\Schedule;
class Bb extends Model {
    use Prunable;
    public $prunableChunkSize = 500;
    . . .
```

```
    public function prunable() {
        return static::whereDate('created_at', '<', now()->subYears(10));
    }

    protected function schedule(Schedule $schedule) {
        $schedule->command('model:prune')->yearly();
    }
}
```

Команда утилиты artisan, выполняющая очистку, имеет следующий формат запуска:

```
php artisan model:prune [--model=<пути к классам моделей>]|↳
[--except=<пути к классам моделей>] [--chunk=<размер чанка>] [--pretend]
```

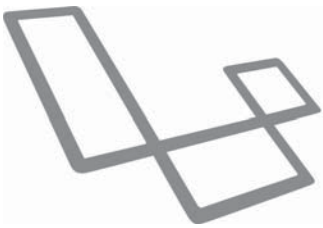
По умолчанию выполняется очистка всех моделей, существующих в проекте.

Поддерживаются следующие командные ключи:

- `--model` — производить очистку только моделей, *пути к классам* которых были указаны. Можно указать произвольное количество *путей к классам моделей*, приведя их через запятую. Пример:

  ```
  php artisan model:prune --model=App\Models\Bb,App\Models\Offer
  ```

- `--except` — очищать все модели, кроме тех, *пути к классам* которых были приведены через запятую;
- `--chunk` — размер чанка по умолчанию, измеряемый в записях (если не задан, принимается значение, заданное в классе модели);
- `--pretend` — выводит количество записей, которые будут удалены в процессе очистки в каждой из моделей, но не удаляет их. Применяется при отладке.



ГЛАВА 17

Шаблоны: расширенные инструменты и дополнительные библиотеки

17.1. Библиотека Laravel HTML: создание веб-форм и элементов управления

Библиотека Laravel HTML упрощает создание некоторых элементов страниц: веб-форм, элементов управления и гиперссылок. Ее установка выполняется командой:

```
composer require laravelcollective/html
```

При использовании этой библиотеки упомянутые ранее элементы страницы создаются не непосредственным написанием их HTML-тегов, а вызовом удобных статических методов класса `Form` и функций.

> ***Полная документация по Laravel HTML...***
> ...находится по адресу: **https://laravelcollective.com/docs/6.x/html**.

17.1.1. Создание элементов управления

Элементы управления создаются вызовами следующих статических методов:

- `label()` — создает надпись с заданным *текстом*, относящуюся к элементу управления с указанным *наименованием*:

  ```
  label(<наименование>, <текст надписи>[, <массив атрибутов тега>=[][,
        <преобразовывать недопустимые символы?>=true]])
  ```

  В ассоциативном *массиве атрибутов тега* ключи элементов должны соответствовать атрибутам тега, а значения элементов зададут значения для этих атрибутов. Пример:

  ```
  {{ Form::label('title', 'Товар') }}
  ```

  По умолчанию все недопустимые символы, содержащиеся в тексте надписи, будут преобразованы в литералы HTML. Для отмены этого преобразования (что может понадобиться при форматировании надписи с помощью HTML-тегов) следует дать параметру *преобразовывать недопустимые символы* значение `false`. Пример:

  ```
  {{ Form::label('title', '<strong>T</strong>овар', [], false) }}
  ```

- `text()` — создает обычное текстовое поле с заданным *наименованием* и заносит в него указанное *изначальное значение*:

```
text(<наименование>[, <изначальное значение>=null[,
                     <массив атрибутов тега>=[]]])
```

Если в серверной сессии присутствует значение, ранее введенное в этот элемент, оно будет выведено вместо *изначального значения* (так что подставлять в вызов метода `text()` функцию `old()` не потребуется). *Массив атрибутов тега* задается в том же формате, что и у метода `label()`. Пример:

```
{{ Form::text('title', $bb->title,
              ['class' => 'form-control' . (($errors->has('title')) ?
                          ' is-invalid' : '')]) }}
```

Следующие методы имеют тот же формат вызова, что и метод `text()`:

- `textarea()` — создает область редактирования;
- `number()` — создает поле ввода целого числа;
- `date()` — создает поле ввода даты;
- `datetime()` — создает поле ввода значения даты и времени;
- `datetimeLocal()` — создает поле ввода значения местных даты и времени;
- `time()` — создает поле ввода времени;
- `month()` — создает поле выбора месяца;
- `week()` — создает поле выбора недели;
- `email()` — создает поле ввода адреса электронной почты;
- `tel()` — создает поле ввода телефонного номера;
- `url()` — создает поле ввода интернет-адреса;
- `search()` — создает поле ввода ключевого слова для поиска;
- `password()` — создает поле ввода пароля;
- `range()` — создает регулятор;
- `color()` — создает поле выбора цвета;
- `file(<наименование>[, <массив атрибутов тега>=[]])` — создает поле для выбора файла (файлов);
- `hidden()` — создает скрытое поле;

Еще методы:

- `checkbox()` — создает флажок с заданными *наименованием*, *значением* и *состоянием*:

```
checkbox(<наименование>[, <значение>=1[, <состояние>=null[,
                          <массив атрибутов тега>=[]]]])
```

Значение будет записано в атрибут `value` тега `<input>`, создающего флажок. *Состояние* должно представлять собой логическую величину: `true` сделает флажок изначально установленным, `false` — сброшенным. Пример:

```
{{ Form::checkbox('publish', 1, $bb->publish,
                  ['class' => 'form-check-input']) }}
```

- `radio()` — создает переключатель с заданными *наименованием*, *значением* и *состоянием*:

```
radio(<наименование>[, <значение>=null[, <состояние>=null[,
                       <массив атрибутов тега>=[]]]])
```

Если заданные *значение* и *состояние* равны, переключатель будет изначально установлен, в противном случае — сброшен. Следует помнить, для сравнивания этих величин используется оператор «строго равно» (===), поэтому они должны принадлежать одному типу. Пример:

```
{{ Form::radio('kind', 'buy', $bb->kind) }}  Купить
{{ Form::radio('kind', 'sell', $bb->kind) }} Продать
```

- `select()` — создает список с заданным *наименованием* на основе указанного *массива пунктов* и делает пункт с заданным *значением* изначально выбранным:

```
select(<наименование>[, <массив пунктов>=[][, <значение>=null[,
       <массив атрибутов списка>=[][, <массив атрибутов пунктов>=[][,
       <массив атрибутов групп пунктов>=[]]]]]])
```

Ассоциативный *массив пунктов* должен содержать элементы одного из двух форматов:

- *<значение пункта> => <текст пункта>*:

```
{{ Form::select('rubric_id', [2 => 'Дома', 3 => 'Гаражи',
                              5 => 'Легковой', 6 => 'Грузовой']) }}
```

- *<заголовок группы> => <массив пунктов, входящих в группу>* — где элементы *массива пунктов* задаются в описанном ранее формате:

```
{{ Form::select('rubric_id',
         ['Здания'    => [2 => 'Дома', 3 => 'Гаражи'],
          'Транспорт' => [5 => 'Легковой', 6 => 'Грузовой']],
      null, ['size' => 6]) }}
```

Ассоциативные *массивы атрибутов списка*, *пунктов* и *групп пунктов* задают атрибуты, привязываемые к тегам самого списка, его пунктов и групп пунктов соответственно, и указываются в том же формате, что и у метода `label()`.

Пример вывода списка рубрик второго уровня, которые разбиты на группы, соответствующие группам первого уровня:

```
public function create() {
    $rubrics = Rubric::whereNull('parent_id')->orderBy('name')->get()
                    ->flatMap(function ($superrubric, $index) {
            return [$superrubric->name => $superrubric->rubrics()
                                ->orderBy('name')->pluck('name', 'id')];
        });
    return view('bb-create', ['rubrics' => $rubrics]);
}
. . .
{{ Form::select('rubric_id', $rubrics) }}
```

- `selectRange()` — создает список для выбора числа в диапазоне от *начального* до *конечного* и делает пункт с числом, равным заданному *значению*, изначально выбранным:

```
selectRange(<наименование>, <начальное число>, <конечное число>[,
            <значение>=null[, <массив атрибутов списка>=[]]])
```

  Пример:

```
{{ Form::selectRange('number', 10, 15, 12) }}
```

- `selectYear()` — создает список для выбора года. Формат вызова такой же, как и у метода `selectRange()`;
- `submit([<надпись>=null[, <массив атрибутов тега>=null]])` — создает кнопку отправки данных с заданной *надписью*:

```
{{ Form::submit('Добавить', ['class' => 'btn btn-primary']) }}
```

  Если *надпись* не указана, будет выведена надпись по умолчанию;

- `image()` — создает графическую кнопку отправки данных, выводящую изображение с заданным *интернет-адресом*:

```
image(<интернет-адрес>[, <наименование>=null[,
                         <массив атрибутов тега>=[]]])
```

  Пример:

```
{{ Form::image('/images/buttons/submit.gif') }}
```

- `reset(<надпись>[, <массив атрибутов тега>=null])` — создает кнопку сброса веб-формы с заданной *надписью*;
- `button()` — создает обычную кнопку с заданной *надписью*. Формат вызова такой же, как и у метода `submit()`;
- `datalist(<якорь>[, <массив пунктов>=[]])` — создает список автозаполнения с заданным *якорем* (записывается в атрибуте тега `id`) на основе заданного массива *пунктов*. Массив пунктов может быть как индексированным:

```
{{ Form::text('title', $bb->title, ['list' => 'idTitle']) }}
{{ Form::datalist('idTitle', ['Дом', 'Гараж']) }}
```

  так и ассоциативным. В последнем случае ключи элементов зададут значения пунктов, заносящиеся в связанное поле ввода, а значения элементов — пояснения, отображаемые в списке ниже значений пунктов. Пример:

```
{{ Form::datalist('idTitle',
        ['Дом' => 'Отдельно стоящий', 'Гараж' => 'Для одной машины']) }}
```

17.1.2. Создание веб-форм

Для создания веб-форм класс `Form` предусматривает три статических метода:

- `open([<массив параметров>=[]])` — создает открывающий тег `<form>`, формирующий веб-форму с параметрами из указанного *массива*. Ассоциативный *массив параметров* может включать элементы со следующими ключами:

- *<имя атрибута тега>* — создает в теге `<form>` одноименный атрибут и заносит в него значение этого элемента:

```
{{ Form::open(['method' => 'GET']) }}
```

  Если у веб-формы указан метод отправки данных, отличный от GET и POST, в форму будет помещено скрытое поле с названием метода, вставленное с применением директивы шаблонизатора `@method` (так что указывать эту директиву вручную не придется). Если метод отправки данных не указан, будет применен метод по умолчанию: POST;

- `url` — задает интернет-адрес для отправки данных из веб-формы:

```
{{ Form::open(['url' => '/rubrics/']) }}
```

- `route` — задает имя маршрута, по которому будут отправлены данные из веб-формы (соответствующий маршруту интернет-адрес будет сгенерирован автоматически):

```
{{ Form::open(['route' => 'rubrics.store']) }}
```

  Если маршрут с заданным именем является параметризованным, значением элемента должен быть массив со следующими элементами:

  - первый — строкового типа — имя маршрута;
  - последующие — формата *<имя URL-параметра> => <значение URL-параметра>* — значения URL-параметров, присутствующих в маршруте.

  Пример:

```
{{ Form::open(['method' => 'PATCH',
        'route' => ['rubrics.update', 'rubric' => $rubric->id]]) }}
```

  Элементы вложенного массива, задающие значения URL-параметров, также могут быть индексированными. Только в этом случае они должны располагаться в том порядке, в котором в шаблонном пути маршрута указаны соответствующие им URL-параметры. Пример:

```
{{ Form::open(['method' => 'PATCH',
               'route' => ['rubrics.update', $rubric->id]]) }}
```

- `action` — задает действие контроллера, которому будут отправлены данные из веб-формы, в виде строки формата *<путь к контроллеру-классу>@<имя действия>*.

  Фреймворк найдет первый маршрут, указывающий на это действие, и самостоятельно сгенерирует на его основе интернет-адрес. Пример:

```
{{ Form::open(['action' =>
               'App\\Http\\Controllers\\RubricController@store']) }}
```

  Если маршрут, указывающий на заданное действие, является параметризованным, значения URL-параметров задаются способом, описанным ранее:

```
{{ Form::open(['method' => 'PATCH',
               'action' => ['App\\Http\\Controllers\\' .
                                'RubricController@update',
                            'rubric' => $rubric->id]]) }}
```

- `files` — со значением `true` — должен быть указан, если веб-форма отправляет файлы.

Помимо открывающего тега `<form>` (и скрытого поля с указанием метода отправки данных, если метод отличен от GET и POST), метод помещает в веб-форму скрытое поле с электронным жетоном безопасности, формируемым директивой шаблонизатора `@csrf`. Так что вручную писать эту директиву не нужно;

- `close()` — создает закрывающий тег веб-формы `</form>`:

```
{{ Form::open(['route' => 'rubrics.store']) }}
    <div class="form-group">
        {{ Form::label('name', 'Название') }}
        {{ Form::text('name', $rubric->name,
                              ['class' => 'form-control']) }}
    </div>
    {{ Form::submit('Добавить', ['class' => 'btn btn-primary']) }}
{{ Form::close() }}
```

- `model(<запись>[, <массив параметров>=[]])` — то же самое, что и `open()`, только дополнительно указывает всем элементам управления, присутствующим в создаваемой веб-форме, брать изначальные значения из заданной *записи*:

```
{{ Form::model($rubric, ['route' => 'rubrics.store']) }}
    <div class="mb-3">
        {{ Form::label('name', 'Название') }}
        {{ Form::text('name', null, ['class' => 'form-control']) }}
    </div>
    {{ Form::submit('Добавить', ['class' => 'btn btn-primary']) }}
{{ Form::close() }}
```

В *разд. 5.6.2* описывались акцессоры — методы моделей, вызываемые при попытке извлечь значения определенных полей и преобразующие эти значения в заданный формат. Есть возможность объявить специальный акцессор, который будет вызываться перед передачей значения поля модели веб-форме, сгенерированной методом `model()`. Он объявляется так же, как обычный акцессор, только его имя должно соответствовать формату `form<имя поля PascalCase>Attribute`. Пример:

```
use Illuminate\Support\Str;
class Rubric extends Model {
    . . .
    public function formNameAttribute($value) {
        return Str::lower($value);
    }
}
```

17.1.3. Создание гиперссылок

Для создания гиперссылок служат следующие функции:

- `link_to()` — создает гиперссылку с заданными *интернет-адресом* и *текстом*:

```
link_to(<интернет-адрес>[, <текст>=null][,
        <массив атрибутов тега>=[][, <HTTPS?>=null[,
        <преобразовывать недопустимые символы>=true]]]])
```

Если *текст* не указан, вместо него будет выведен *интернет-адрес*. *Массив атрибутов тега* задается в том же формате, что и у метода `label()` (см. *разд. 17.1.1*). Если параметру *HTTPS* дать значение `true`, будет сгенерирован интернет-адрес, использующий протокол HTTPS, если дать значение `false` — интернет-адрес с протоколом HTTP, а если `null` — интернет-адрес с текущим протоколом. Пример:

```
{{ link_to('/', 'На главную') }}
```

По умолчанию все недопустимые символы в тексте гиперссылки будут преобразованы в литералы HTML. Для отмены этого преобразования (что может понадобиться при выводе текста, содержащего HTML-теги) следует дать параметру *преобразовывать недопустимые символы* значение `false`. Пример:

```
{{ link_to('/', '<em>На главную</em>', [], null, false) }}
```

- `link_to_route()` — создает гиперссылку на маршрут с заданным *именем*, имеющую указанный *текст*:

```
link_to_route(<имя маршрута>[, <текст>=null][,
              <массив URL-параметров>=[][,
              <массив атрибутов тега>=[]]]])
```

Массив URL-параметров должен содержать элементы, хранящие значения соответствующих URL-параметров и выстроенные в порядке следования этих параметров в шаблонном пути. Также можно указывать в нем элементы формата *<имя URL-параметра> => <значение URL-параметра>*. Примеры:

```
{{ link_to_route('index', 'На главную') }}
{{ link_to_route('rubric', 'Гаражи', ['rubric' => 3]) }}
```

- `link_to_action()` — создает гиперссылку на заданное *действие контроллера*, имеющую указанный *текст*:

```
link_to_action(<действие контроллера>[, <текст>=null][,
               <массив URL-параметров>=[][,
               <массив атрибутов тега>=[]]]])
```

Действие контроллера можно указать в виде:

  - строки формата *<путь к контроллеру-классу>@<имя действия>*:

```
{{ link_to_action(
           'App\\Http\\Controllers\\MainController@rubric',
           'Гаражи', [3]) }}
```

  - массива из двух строковых элементов — пути к контроллеру-классу и имени действия:

```
{{ link_to_action(
        [App\Http\Controllers\MainController::class, 'rubric'],
        'Гаражи', [3]) }}
```

- `link_to_asset()` — создает гиперссылку на статический файл с указанным *интернет-адресом* и *текстом*, в остальном аналогичен методу `link_to()`:

```
link_to_asset(<интернет-адрес>[, <текст>=null][,
              <массив атрибутов тега>=[][, <HTTPS?>=null]]])
```

Пример:

```
{{ link_to_asset('/archives/price.zip', 'Прайс-лист') }}
```

17.2. Библиотека genert/bbcode: поддержка BBCode

BBCode (Bulletin Board Code, код досок объявлений) — это язык разметки, который используется для форматирования текста на многих форумах и блогах. Форматирование выполняется с помощью тегов, схожих с тегами языка HTML, но заключаемых в квадратные скобки. При выводе такие теги преобразуются в обычный HTML-код.

Для обработки тегов BBCode в Laravel удобно применять библиотеку genert/bbcode. Она устанавливается командой:

```
composer require genert/bbcode
```

После чего необходимо открыть модуль config\app.php и:

- в список зарегистрированных в проекте провайдеров `providers` — добавить провайдер `BBCodeServiceProvider`:

```
'providers' => [
    . . .
    Genert\BBCode\BBCodeServiceProvider::class,
],
```

- в список зарегистрированных обозначений фасадов `aliases` — добавить фасад `BBCode`, дав ему одноименное обозначение:

```
'aliases' => Facade::defaultAliases()->merge([
    . . .
    'BBCode' => Genert\BBCode\Facades\BBCode::class,
])->toArray(),
```

Более подробно списки провайдеров и фасадов будут рассмотрены в *главе 20*.

> ***Полная документация по genert/bbcode...***
> ...находится по адресу: **https://github.com/Genert/BBCode**.

17.2.1. Использование библиотеки genert/bbcode

Для преобразования текста, размеченного BBCode-тегами, в HTML-код применяются следующие методы фасада `Genert\BBCode\Facades\BBCode`:

- `convertToHtml(<текст>)` — преобразует *текст*, размеченный BBCode-тегами, в HTML-код и возвращает его в качестве результата.

> ***Для вывода используйте директиву {!! ... !!}!***
> Поскольку директива `{{ ... }}` выведет HTML-код как есть.

Пример:

```
{!! BBCode::convertToHtml($bb->content) !!}
```

- `stripBBCodeTags(<текст>)` — удаляет из *текста* все BBCode-теги и возвращает результат;
- `addLinebreakParser()` — активизирует преобразование последовательностей символов возврата каретки и перевода строки (`\r\n`) в HTML-теги `<br>`. Вызов этого метода следует поместить в метод `boot()` провайдера `AppServiceProvider` или любого другого — это гарантирует, что метод будет вызван перед выводом первой страницы. Пример:

```
use Genert\BBCode\Facades\BBCode;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        BBCode::addLinebreakParser();
    }
}
```

- `only()` — предписывает библиотеке обрабатывать только BBCode-теги с указанными *обозначениями* (обозначения тегов будут приведены далее). Поддерживает два формата вызова:

```
only(<обозначение 1>, <обозначение 2>, ... <обозначение n>)
only(<массив обозначений тегов>)
```

  В качестве результата возвращается текущий объект обработчика BBCode-тегов, что позволяет записывать вызовы этого метода цепочкой.

  Чтобы включить обработку только определенных BBCode-тегов во всем проекте, следует поместить вызов метода `only()` в тело метода `boot()` провайдера `AppServiceProvider` перед вызовом метода `addLinebreakParser()`:

```
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        BBCode::only('bold', 'italic')->addLinebreakParser();
    }
}
```

- `except()` — предписывает библиотеке обрабатывать все BBCode-теги за исключением тегов с указанными *наименованиями*. Форматы вызова аналогичны таковым у метода `only()`.

17.2.2. Поддерживаемые BBCode-теги

Все BBCode-теги, поддерживаемые библиотекой genert/bbcode, вместе с их обозначениями приведены в табл. 17.1.

Таблица 17.1. *BBCode-теги, поддерживаемые библиотекой genert/bbcode*

BBCode-тег	Описание	Обозначение
`[b]<текст>[/b]`	**Полужирный** текст	`bold`
`[i]<текст>[/i]`	*Курсивный* текст	`italic`

Таблица 17.1 *(окончание)*

BBCode-тег	Описание	Обозначение
[u]<*текст*>[/u]	Подчеркнутый текст	underline
[s]<*текст*>[/s]	~~Зачеркнутый~~ текст	strikethrough
[sub]<*текст*>[/sub]	Верхний индекс	sub
[sup]<*текст*>[/sup]	Нижний индекс	sup
[small]<*текст*>[/small]	Текст, набранный уменьшенным шрифтом	small
[img]<*адрес*>[/img]	Графическое изображение с заданным *адресом*	image
[youtube]<*код*>[/youtube]	Видео с YouTube. Код *видео* берется из GET-параметра v его интернет-адреса	youtube
[url]<*адрес*>[/url]	Гиперссылка. В качестве текста подставляется *адрес*	link
[url=<*адрес*>]<*текст*>[/url]	Гиперссылка с заданным *текстом*	namedlink
[h1]<*текст*>[/h1]	Заголовок первого уровня	h1
[h2]<*текст*>[/h2]	Заголовок второго уровня	h2
[h3]<*текст*>[/h3]	Заголовок третьего уровня	h3
[h4]<*текст*>[/h4]	Заголовок четвертого уровня	h4
[h5]<*текст*>[/h5]	Заголовок пятого уровня	h5
[h6]<*текст*>[/h6]	Заголовок шестого уровня	h6
[quote]<*текст*>[/quote]	Блочная цитата	quote
[list=1]<*пункты*>[/list]	Нумерованный список с нумерацией в виде арабских цифр	orderedlistnumerical
[list=a]<*пункты*>[/list]	Нумерованный список с нумерацией в виде латинских букв	orderedlistalpha
[list]<*пункты*>[/list]	Маркированный список	unorderedlist
[*]<*текст*>	Пункт списка	listitem
[code]<*текст*>[/code]	Текст, набранный моноширинным шрифтом	code
[table]<*строки*>[/table]	Таблица	table
[tr]<*ячейки*>[/tr]	Строка таблицы	table-row
[td]<*текст*>[/td]	Ячейка таблицы	table-data

17.2.3. Добавление своих BBCode-тегов

Добавить библиотеке genert/bbcode поддержку своих BBCode-тегов можно вызовом у фасада `BBCode` метода `addParser()`:

`addParser(<`*обозначение тега*`>, <`*шаблон*`>, <`*замена*`>, <`*содержимое*`>)`

Обозначение добавляемого тега не должно совпадать с обозначениями уже имеющихся тегов (см. табл. 17.1). *Шаблон* должен представлять собой регулярное выражение, в котором для извлечения фрагментов содержимого используются группы. В *замене* и *содержимом* для вставки фрагментов содержимого следует использовать обратные ссылки формата `$<порядковый номер группы>`, где нумерация групп начинается с 1. *Содержимое* будет подставляться вместо тега при обработке текста методом `stripBBCodeTags()`.

В качестве результата метод возвращает текущий объект обработчика BBCode-тегов, что позволяет записывать цепочки его вызовов.

Вызов метода `addParser()` также лучше поместить в метод `boot()` провайдера `AppServiceProvider` или какого-либо другого перед вызовом метода `addLinebreakParser()`.

Пример добавления нового тега `[align=left|center|right]<текст>[/align]`, задающего выравнивание *текста*:

```
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        BBCode::addParser('align',
                    '/\[align\=(left|center|right)\](.*?)\[\/align\]/s',
                    '<div style="text-align: $1;">$2</div>', '$2')
                ->addLinebreakParser();
    }
}
```

17.3. Библиотека Captcha for Laravel: поддержка CAPTCHA

Если планируется дать посетителям-гостям возможность добавлять какие-либо данные в базу (например, оставлять комментарии), не помешает как-то обезопасить сайт от программ рассылки спама. Одно из решений — применение CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart, полностью автоматизированный публичный тест Тьюринга для различения компьютеров и людей).

CAPTCHA выводится на веб-страницу в виде графического изображения, содержащего сильно искаженный или зашумленный текст, который нужно прочитать и занести в расположенное рядом поле ввода. Если результат оказался верным, то, скорее всего, данные занесены человеком, поскольку программам такие сложные задачи пока еще не по плечу.

Captcha for Laravel — одна из библиотек, реализующих поддержку CAPTCHA.

> ***Для успешной работы Captcha for Laravel…***
>
> …в PHP следует активизировать расширение GD2. Без этого расширения библиотеку даже не удастся установить.

Библиотека устанавливается командой:

```
composer require mews/captcha
```

После этого необходимо открыть модуль config\app.php и добавить в список зарегистрированных в проекте провайдеров `CaptchaServiceProvider`:

```
'providers' => [
    . . .
    Mews\Captcha\CaptchaServiceProvider::class,
],
```

Если для вывода CAPTCHA планируется использовать фасад `Captcha`, его следует добавить в список фасадов:

```
'aliases' => Facade::defaultAliases()->merge([
    . . .
    'Captcha' => Mews\Captcha\Facades\Captcha::class,
])->toArray(),
```

> ***Полная документация по Captcha for Laravel...***
> ...находится по адресу: **https://github.com/mewebstudio/captcha**.

17.3.1. Настройка Captcha for Laravel

Настройки библиотеки записываются в модуле config\captcha.php. Изначально он отсутствует в папке config, и, чтобы создать его, следует набрать команду:

```
php artisan vendor:publish --provider=Mews\Captcha\CaptchaServiceProvider
```

Параметр `characters` хранит массив символов, которые будут использоваться в CAPTCHA. Изначально он содержит цифры от 2 до 9 (цифры 1 нет, поскольку она похожа на латинскую «l»), а также строчные и прописные буквы латиницы (за исключением букв «i», «l», «o», «s», «v» и «w», которые похожи либо на цифры, либо на другие буквы).

Остальной код модуля config\captcha.php создает *пресеты* (предварительно созданные конфигурации, имеющие уникальные имена) CAPTCHA. Изначально их пять: `default` (используется по умолчанию, если при выводе CAPTCHA не было указано имя), `math`, `flat`, `mini` и `inverse`, также можно добавить свои собственные.

Вот параметры, которые можно указать в пресетах:

- `length` — длина строки CAPTCHA в символах (по умолчанию: 5);
- `sensitive` — если `false`, символы будут выводиться только в нижнем регистре, если `true` — в нижнем и верхнем регистрах (по умолчанию: `false`);
- `width` — ширина CAPTCHA в пикселах (по умолчанию: 120);
- `height` — высота CAPTCHA в пикселах (по умолчанию: 36);
- `lines` — количество прямых линий случайной длины, проводимых на изображении под случайными углами, чтобы усложнить обработку программами-роботами (по умолчанию: 3);
- `angle` — максимальный угол наклона символов в градусах. Символы будут выводиться наклоненными под случайно выбранным углом в диапазоне от `-<angle>` до `<angle>`. По умолчанию: 15;

- `fontColors` — массив цветов, которыми будут выведены отдельные символы CAPTCHA, в стандарте CSS в виде строк. Цвета из этого массива будут выбираться случайным образом. По умолчанию: «пустой» массив (цвета выбираются самой библиотекой произвольно);
- `fontsDirectory` — путь к папке со шрифтами, которыми будут выводиться символы. Если параметр не указан, будут использованы шрифты, входящие в состав библиотеки и хранящиеся в папке vendor\mews\captcha\assets\fonts;
- `bgImage` — если `true`, CAPTCHA будет содержать случайно выбранное фоновое изображение (что усложняет обработку программами-роботами), если `false` — не будет содержать (по умолчанию: `true`). Фоновые изображения будут взяты из папки vendor\mews\captcha\assets\backgrounds;
- `bgColor` — фоновый цвет, который CAPTCHA будет иметь, если фоновое изображение отключено, в стандарте CSS в виде строки (по умолчанию: `'#ffffff'`, т. е. белый);
- `textLeftPadding` — отступ между левой границей изображения и текстом в пикселах (по умолчанию: 4);
- `quality` — относительное качество изображения CAPTCHA от 1 до 100 (по умолчанию: 90);
- `contrast` — уровень контрастности от –100 до 100, отрицательные величины уменьшают контрастность, положительные — увеличивают (по умолчанию: 0, т. е. контрастность не изменяется);
- `sharpen` — уровень резкости изображения от 0 (резкость не изменяется) до 100 (по умолчанию: 0);
- `blur` — уровень размытия изображения от 0 (размытие не изменяется) до 100 (по умолчанию: 0);
- `invert` — если `true`, цвета изображения будут инвертированы, если `false` — не будут (по умолчанию: `false`);
- `math` — если `false`, будет выведен обычный CAPTCHA в виде строки случайных символов. Если `true`, выводится математический CAPTCHA, представляющий собой математическое выражение, результат вычисления которого нужно ввести. По умолчанию: `false`.

Пример указания пресетов `default` и `mini`:

```
'default' => ['length' => 6, 'width' => 200, 'height' => 50,
                                              'quality' => 90],
'mini' =>    ['length' => 3, 'width' => 70, 'height' => 32],
```

17.3.2. Использование Captcha for Laravel

Чтобы защитить веб-форму с применением CAPTCHA, следует вывести в ней саму CAPTCHA и поле ввода для занесения показанного на ней текста. Поле ввода может иметь произвольное наименование (сами разработчики библиотеки рекомендуют давать наименование `captcha`).

Для вывода CAPTCHA применяются функции:

- `captcha_img()` — возвращает HTML-код тега `<img>`, выводящего на страницу изображение CAPTCHA:

  `captcha_img([<пресет>='default'[, <массив атрибутов тега>=[]]])`

  В ассоциативном *массиве атрибутов тега* `<img>` ключи элементов должны соответствовать атрибутам тега `<img>`, выводящего CAPTCHA, а значения элементов зададут значения для этих атрибутов.

  > ***Для вывода используйте директиву `{!! ... !!}`!***
  > Поскольку директива `{{ ... }}` выведет HTML-код как есть.

  Пример:

```
<div>{!! captcha_img() !!}</div>
<div><input name="captcha"></div>
```

  Вместо этой функции можно использовать метод `img()` фасада `Captcha`, имеющий тот же формат вызова:

```
<div>{!! Captcha::img('mini', ['class' => 'form-captcha']) !!}</div>
```

- `captcha_src([<пресет>='default'])` — возвращает интернет-адрес изображения CAPTCHA.

  Любители фасадов могут использовать метод `src()` фасада `Captcha`, имеющий тот же формат вызова:

```
<div><img src="{{ Captcha::src() }}" class="form-captcha"></div>
```

Для проверки правильности ввода текста с CAPTCHA следует выполнить валидацию поля ввода, в которое заносится этот текст, с применением правил `required` и `captcha` (о валидации рассказывалось в *главе 10*). Пример:

```
public function store(Request $request) {
    $validationRules = ['captcha' => 'required|captcha'];
    $errorMessages = [
        'captcha.required' => 'Введите текст с картинки',
        'captcha.captcha' => 'Введите правильный текст с картинки'
    ];
    $validated = $request->validate($validationRules, $errorMessages);
    . . .
}
```

17.4. Написание своих директив шаблонизатора

Код, объявляющий новые директивы шаблонизатора Laravel, пишется в теле метода `boot()` провайдера `AppServiceProvider` или любого другого.

17.4.1. Написание простейших директив

Для объявления простейшей директивы шаблонизатора, принимающей один параметр, применяется метод `directive()` фасада `Illuminate\Support\Facades\Blade`:

```
directive(<имя директивы>, <анонимная функция>)
```

Имя директивы должно быть уникальным, содержать лишь буквы латиницы, цифры и символы подчеркивания. *Анонимная функция* должна принимать параметр, передаваемый директиве, и возвращать PHP-код, который реализует эту директиву и будет вставлен в код откомпилированного шаблона.

Пример объявления директивы, выводящей значение временно́й отметки в формате *<число>.<месяц>.<год> <часы>:<минуты>*:

```
use Illuminate\Support\Facades\Blade;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('d.m.Y H:i'); ?>";
        });
    }
}
```

После объявления новой директивы ее можно использовать в шаблонах, записав конструкцию формата `@<имя директивы>(<значение параметра>)`, например:

```
<p>Дата и время публикации: @datetime($bb->created_at)</p>
```

> ***ПОСЛЕ ПРАВКИ КОДА НОВОЙ ДИРЕКТИВЫ СЛЕДУЕТ УДАЛИТЬ ОТКОМПИЛИРОВАННЫЕ ШАБЛОНЫ...***
>
> ...поскольку шаблонизатор Laravel не отслеживает правку провайдеров, в которых объявляются директивы, и соответственно не перекомпилирует шаблоны. Для удаления откомпилированных шаблонов следует набрать команду:
>
> ```
> php artisan view:clear
> ```

Можно создавать директивы, не принимающие параметра, — для этого в вызове метода `directive()` следует указать анонимную функцию без параметров. Вот пример директивы, выводящей имя текущего пользователя или «пустую» строку, если вход не был выполнен:

```
public function boot() {
    . . .
    Blade::directive('username', function () {
        return "<?php echo Auth::check() ? Auth::user()->name : ''; ?>";
    });
}
```

Пример использования этой директивы:

```
<span>Текущий пользователь: @username</span>
```

17.4.1.1. Форматировщики объектов

Форматировщик объекта преобразует в заданный строковый вид объект указанного класса при попытке вывести его на страницу. Такие форматировщики можно рассматривать как более удобную альтернативу директивам шаблонизатора.

Форматировщик создается вызовом метода `stringable(<анонимная функция>)` фасада `Blade`, который записывается в теле метода `boot()` провайдера `AppServiceProvider` (или любого другого). Задаваемая *анонимная функция* должна принимать в качестве параметра выводимый объект, у которого в качестве типа указан нужный класс, и возвращать его строковое представление в необходимом формате.

Пример двух форматировщиков объектов, один из которых выводит временну́ю отметку, представленную объектом класса `Carbon`, а второй — имя пользователя, представленного моделью `User`:

```
use Illuminate\Support\Carbon;
use App\Models\User;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Blade::stringable(function (Carbon $datetime) {
            return $datetime->format('d.m.Y H:i');
        });

        Blade::stringable(function (User $user) {
            return $user->name;
        });
    }
}
```

Чтобы задействовать эти форматировщики объектов, достаточно выполнить вывод объектов указанных в них классов с помощью директив, описанных в *разд. 11.2.1*:

```
<p>Дата и время публикации: {{ $bb->created_at }}</p>
<p>Текущий пользователь: {{ Auth::user() }}</p>
```

17.4.2. Написание условных директив

Для написания условных директив, выводящих заданный фрагмент HTML-кода, если выполняется указанное условие, фасад `Blade` предлагает метод `if()`, формат вызова которого совпадает с таковым у метода `directive()` (см. *разд. 17.4.1*). Задаваемая в вызове этого метода *анонимная функция* должна возвращать логическую величину: `true`, если реализуемое директивой условие выполняется, и `false` — в противном случае. Как правило, такие директивы принимают параметр и проверяют выполнение заданного условия на основе его значения.

Пример объявления директивы `published`, проверяющей, помечено ли переданное ей объявление как предназначенное к публикации:

```
public function boot() {
    . . .
    Blade::if('published', function ($bb) {
        return $bb->publish;
    });
}
```

После объявления условной директивы в коде шаблона можно использовать следующие директивы:

- @<*имя директивы*>(<*значение 1*>)
  <*содержимое if 1 — выводится, если* значение 1 *удовлетворяет условию*>
  @else<*имя директивы*>(<*значение 2*>)
  <*содержимое if 2 — выводится, если* значение 2 *удовлетворяет условию*>
  . . .
  [@else
  <*содержимое else — выводится, если ни одно* значение *не удовлетворяет условию*>]
  @end<*имя директивы*>

  Пример:

  ```
  @published($bb)
  <p>Объявление опубликовано</p>
  @else
  <p>Объявление не опубликовано</p>
  @endpublished
  ```

- @unless<*имя директивы*>(<*значение*>)
  <*содержимое unless — выводится, если* значение *не удовлетворяет условию*>
  @end<*имя директивы*>

  Пример:

  ```
  @unlesspublished($bb)
  <p>Объявление не опубликовано</p>
  @endpublished
  ```

17.5. Пакет Laravel Mix

Разработчики Laravel предлагают инструменты для упрощения верстки страниц сайта — в частности, создания таблиц стилей и веб-сценариев. Они позволяют транслировать таблицы стилей из производных языков (SCSS, LESS, Stylus) в CSS, метить статические файлы и др.

Все эти инструменты сведены в пакет Laravel Mix, написанный на языке JavaScript и работающий под управлением программной среды Node.js, которая должна быть установлена на компьютере.

Установка пакета Laravel Mix и всех необходимых зависимостей выполняется набором в папке проекта команды:

```
npm install
```

Также устанавливаются библиотеки lodash и axios...

...часто используемые при программировании веб-сценариев.

Полная документация по Laravel Mix...

...находится по адресу: **https://laravel-mix.com/docs/6.0**. Это мощный пакет с большим количеством программных инструментов, поэтому имеет смысл потратить время на его изучение.

Полезно знать...

Laravel Mix является надстройкой над популярным пакетом управления веб-проектами Webpack. Последний также устанавливается в числе необходимых зависимостей.

17.5.1. Исходные файлы и их расположение

Для хранения исходных файлов таблиц стилей и веб-сценариев предназначены следующие папки:

- resources\js — веб-сценарии, написанные на обычном JavaScript.

  Изначально там присутствуют следующие файлы:

  - app.js — точка входа проекта. Изначально содержит только команду импорта файла bootstrap.js;
  - bootstrap.js — файл для хранения инициализирующего кода. Изначально содержит команды импорта библиотек lodash и axios.

  Laravel Mix уже сконфигурирован так, чтобы сохранять собранный код в файле public\js\app.js;

- resources\css — таблицы стилей, написанные на языке CSS.

  Изначально там присутствует «пустой» главный файл app.css. Собранный код будет сохранен в файле public\css\app.css.

Созданные таким образом файлы можно привязать к веб-страницам, вставив в базовый шаблон код:

```
<head>
    . . .
    <link rel="stylesheet" href="/css/app.css">
    <script src="/js/app.js"></script>
</head>
```

Можно создать в этих папках новые таблицы стилей и веб-сценарии, равно как и создать новые папки для хранения таблиц стилей, написанных на SCSS, LESS, Stylus, графических изображений и пр. Это может понадобиться при написании сайтов со сложным оформлением. Однако тогда потребуется внести в конфигурацию Laravel Mix необходимые правки, указав, в частности, где сохранять результаты обработки исходных файлов.

17.5.2. Конфигурирование Laravel Mix

Конфигурация Laravel Mix сохраняется в файле webpack.mix.js, находящемся непосредственно в папке проекта, и пишется на языке JavaScript. Код конфигурации представля-

ет собой вызовы различных методов объекта программного ядра Laravel Mix, хранящегося в переменной `mix`. Например, изначально файл webpack.mix.js хранит такой код (комментарии удалены):

```
const mix = require('laravel-mix');
mix.js('resources/js/app.js', 'public/js')
   .postCss('resources/css/app.css', 'public/css', []);
```

17.5.2.1. Обработка таблиц стилей

Объект программного ядра Laravel Mix поддерживает методы, приведенные далее. Все они в качестве результата возвращают ссылку на текущий объект, благодаря чему можно записывать их вызовы цепочкой.

- `postCss()` — обрабатывает написанный на обычном CSS *исходный файл* с применением программы PostCSS (**https://postcss.org/**) и сохраняет результат в *конечном файле*:

  ```
  postCss(<исходный файл>, <конечный файл>[, <массив плагинов>])
  ```

  Исходный и *конечный* файлы задаются в виде путей к ним, записанных относительно папки проекта (пример см. ранее). Вместо полного пути к конечному файлу можно задать путь к папке, в которой он должен находиться, — тогда конечный файл получит то же имя, что и начальный. Пример:

  ```
  mix.postCss('resources/css/app.css', 'public/css');
  ```

  Также можно указать *массив плагинов*, которые должны выполняться совместно с PostCSS. Пример:

  ```
  mix.postCss('resources/css/app.css', 'public/css/app.css',
              [require('precss')(),
               require('cssnano')({preset: 'default'})]);
  ```

- `css()` — то же самое, что и `postCss()`;
- `sass()` — транслирует написанный на языке SCSS *исходный файл* таблицы стилей в CSS и сохраняет результат в *конечном файле*:

  ```
  sass(<исходный файл>, <конечный файл>[, <параметры транслятора>])
  ```

  Исходный и *конечный* файлы задаются так же, как и в вызове метода `postCss()`. Пример:

  ```
  mix.sass('resources/sass/app.scss', 'public/css/app.css');
  ```

  Также можно указать *параметры транслятора*. Пример:

  ```
  mix.sass('resources/sass/app.scss', 'public/css/app.css',
                                      {precision: 5});
  ```

  ПАРАМЕТРЫ ТРАНСЛЯТОРОВ ОПИСАНЫ НА ПОСВЯЩЕННЫХ ИМ ВЕБ-САЙТАХ
  Ссылки на эти сайты приведены в документации по Laravel Mix.

- `less()` — выполняет трансляцию таблиц стилей, написанных на языке LESS. В остальном аналогичен `sass()`;
- `stylus()` — выполняет трансляцию таблиц стилей, написанных на языке Stylus. В остальном аналогичен `sass()`;

- `styles()` — просто объединяет указанные в *массиве* исходные файлы таблиц стилей, написанных на CSS, в один *конечный файл*:

  `styles(<массив исходных файлов>, <конечный файл>)`

  Пример:

  ```
  mix.styles(['resources/css/base.css', 'resources/css/layout.css',
              'resources/css/details.css'], 'public/css/styles.css')
  ```

По умолчанию Laravel Mix преобразует все относительные интернет-адреса, записанные в CSS-функциях `url()`, в абсолютные (так, адрес **../imgs/backgrounds/bg2.jpg** будет преобразован в **/imgs/backgrounds/bg2.jpg**). Если это по какой-либо причине неприемлемо (например, когда структура папок в папке public отличается от таковой в папке resources), такое преобразование можно отключить. Для этого следует вызвать метод `options(<объект с параметрами>)`, передав ему простой *объект*, который содержит свойство `processCssUrls` со значением `false`. Пример:

```
mix.sass('resources/sass/app.scss', 'public/css')
   .options({processCssUrls: false});
```

Также можно включить генерирование карт исходного кода (source maps), вызвав метод `sourceMaps()`:

```
mix.sass('resources/sass/app.scss', 'public/css').sourceMaps();
```

17.5.2.2. Обработка веб-сценариев

Для обработки веб-сценариев предназначены методы:

- `js()` — обрабатывает заданный *исходный файл* и сохраняет результат в указанном *конечном файле*. Если указан *массив исходных файлов*, все они будут объединены в один *конечный файл*. Формат вызова:

  `js(<исходный файл>|<массив исходных файлов>, <конечный файл>)`

  В процессе преобразования:

  - исходные файлы, написанные на языке JavaScript стандарта ES2017 с использованием модулей, — транслируются в стандарт JavaScript, «понимаемый» всеми веб-обозревателями, включая устаревшие;
  - неиспользуемый программный код — удаляется.

  Компоненты Vue в обычный JavaScript в новой версии Laravel Mix более не компилируются. Чтобы произвести их компиляцию, следует предпринять дополнительные действия (см. далее).

  Исходные и *конечный* файлы задаются в виде путей к ним, записанных относительно папки проекта. Вместо полного пути к конечному файлу можно задать путь к папке, в которой он должен находиться, — тогда конечный файл получит то же имя, что и начальный. Примеры:

  ```
  mix.js('resources/js/app.js', 'public/js');
  mix.js(['resources/js/lib1.js', 'resources/js/lib2.js',
          'resources/js/lib3.js', 'resources/js/app.js'],
         'public/js/app.js');
  ```

- `react()` — указывает дополнительно откомпилировать все компоненты React в JavaScript. Вызывается после вызова метода `js()`. Пример;

```
mix.js('resources/js/app.js', 'public/js').react();
```

- `vue(<объект с параметрами>)` — указывает дополнительно откомпилировать все компоненты Vue в JavaScript. Вызывается после вызова метода `js()`. Дополнительно можно указать простой *объект с параметрами* транслятора. Пример

```
mix.js('resources/js/app.js', 'public/js').vue();
```

- `scripts()` — просто объединяет указанные в *массиве* исходные файлы веб-сценариев, написанных на JavaScript, в один *конечный файл* без какой бы то ни было обработки:

```
scripts(<массив исходных файлов>, <конечный файл>)
```

Часто при программировании веб-сценариев используются сторонние библиотеки, которые, как правило, редко изменяются и имеют большой объем. При объединении нескольких JavaScript-файлов, хранящих сторонние библиотеки, и файла с кодом, написанным разработчиками сайта, конечный файл получается очень большим. Такие файлы долго загружаются и плохо подходят для долговременного хранения в кеше веб-обозревателя, поскольку даже незначительное изменение в коде сайта вызывает его повторную, опять же, очень долгую загрузку.

Решить проблему можно, вынеся все сторонние библиотеки в один файл, а код веб-сценариев сайта — в другой. Это можно сделать, вызвав метод `extract([<массив библиотек и модулей>])`. В *массиве* указываются имена выделяемых в отдельный файл библиотек и модулей, если же *массив* не указан, в отдельный файл будут выделены все импортированные библиотеки и модули. Пример:

```
mix.js('resources/js/app.js', 'public/js')
   .extract(['jquery', 'popper.js', 'bootstrap', 'lodash', 'axios']);
```

В результате будут созданы три файла: manifest.js (манифест Webpack, управляющий загрузкой кода), vendor.js (код выделенных библиотек) и app.js (код веб-сценариев сайта). Все их нужно привязать к страницам:

```
<head>
    . . .
    <script src="/js/manifest.js"></script>
    <script src="/js/vendor.js"></script>
    <script src="/js/app.js"></script>
</head>
```

17.5.2.3. Копирование файлов и папок

Иногда бывает необходимо просто скопировать из папки resources в папку public какие-либо файлы или целые папки с файлами (например, графические изображения или шрифты). Для этого применяются методы:

- `copy()` — копирует *исходный файл* на *новое местоположение*:

```
copy(<исходный файл>|<массив исходных файлов>, <новое местоположение>)
```

Исходный файл задается в виде пути к нему, отсчитанного от папки проекта. В пути можно использовать литералы: `*` (обозначает произвольную последовательность любых символов, кроме слешей) и `**` (обозначает произвольный фрагмент пути, включая слеши). *Новое местоположение* указывается в виде пути к файлу назначения или пути к папке, где он должен находиться (в этом случае файл будет скопирован под своим изначальным именем). Примеры:

```
mix.copy('resources/images/bg1.jpg', 'public/imgs/bg.jpg');
mix.copy('resources/images/bg2.jpg', 'public/imgs');
mix.copy('resources/archives/*.zip', 'public/archives');
mix.copy('resources/others/**/*.png', 'public/images/others');
```

Также можно указать *массив исходных файлов*:

```
mix.copy(['resources/archives/price.zip',
          'resources/archives/order_form.zip'], 'public/archives');
```

- `copyDirectory(<`*исходная папка*`>, <`*папка назначения*`>)` — копирует все файлы из *исходной папки* в *папку назначения*:

```
mix.copyDirectory('resources/images', 'public/imgs');
```

17.5.2.4. Мечение файлов

Часто случается так, что после изменения какого-либо файла (например, таблицы стилей) веб-обозреватель продолжает использовать его старую редакцию, хранящуюся в кеше («*застревание в кеше*»). Чтобы заставить его загрузить новую редакцию файла, к именам файлов добавляют хеши, вычисленные на основе содержимого этих файлов, — *метят* их.

Мечение файлов выполняет метод `version()`, вызываемый в самом конце цепочки вызовов. Пример его использования:

```
mix.js('resources/js/app.js', 'public/js')
   .sass('resources/sass/app.scss', 'public/css/app.css')
   .version();
```

При этом в папке public создается файл mix-manifest.json, впоследствии используемый Laravel для вставки интернет-адресов меченых файлов в код шаблонов.

Вставить интернет-адрес меченого файла в код шаблона Laravel можно вызовом функции-хелпера `mix(<`*путь*`>)`, где *путь* указывается так, будто файл не был мечен. Пример:

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}">
```

В результате в код сгенерированной страницы будет помещен тег:

```
<link rel="stylesheet" href="/css/app.css?id=b286c2360ac34b141068">
```

где GET-параметр `id` хранит хеш меченого файла (указан хеш, получившийся у автора, у вас он может отличаться).

Если статические файлы будут размещаться на другом веб-сервере, следует указать функции `mix()`, чтобы она генерировала интернет-адрес, указывающий на этот веб-сервер. Для этого в модуль config\app.php нужно добавить настройку `mix_url` и записать в нее префикс интернет-адреса:

```
return [
    . . .
    'mix_url' => 'http://cdn.bboard.ru/static',
]
```

Впрочем, лучше записать этот префикс в файл .env, где хранятся локальные настройки, — так удобнее при работе в составе команды:

```
// Файл .env
MIX_ASSET_URL=http://cdn.bboard.ru/static
. . .
// Модуль config\app.php
return [
    . . .
    'mix_url' => env('MIX_ASSET_URL', null),
]
```

17.5.3. Запуск Laravel Mix

Запустить Laravel Mix для обработки статических файлов можно одной из следующих команд:

- `npm run dev`

  Выполняет единовременную обработку файлов, после чего Laravel Mix завершает работу.

  Скорее всего, при первом выполнении этой команды Laravel Mix будет устанавливать необходимые ему для работы дополнительные библиотеки. Как только он их установит, следует отдать эту же команду еще раз — для собственно обработки файлов;

- `npm run watch`

  Laravel Mix остается запущенным, отслеживает изменение исходных файлов и обрабатывает изменившиеся файлы повторно. Рекомендуется к использованию, т. к. значительно увеличивает производительность.

  Остановить работу Laravel Mix можно нажатием комбинации клавиш <Ctrl>+ +<Break> или <Ctrl>+<C>;

- `npm run watch-poll`

  То же самое. Используется в случае, если предыдущая команда почему-то не обрабатывает изменившиеся файлы.

При использовании последних двух команд для каждого успешного или неуспешного преобразования выводится соответствующее системное уведомление. Есть возможность отключить эти уведомления, использовав методы:

- `disableSuccessNotifications()` — отключает только уведомления об успешном преобразовании:

  ```
  mix.disableSuccessNotifications();
  ```

- `disableNotifications()` — отключает любые уведомления.

17.6. Использование Bootstrap

CSS-фреймворк Bootstrap в настоящее время настолько популярен, что многие веб-фреймворки, и Laravel в том числе, генерируют шаблоны, рассчитанные на его использование. Но Laravel идет дальше, предлагая установить Bootstrap непосредственно в состав проекта и загружать его не со сторонних сервисов, а локально.

Для подготовки к установке Bootstrap в составе проекта следует удостовериться, что библиотека laravel/ui установлена, после чего набрать команду:

```
php artisan ui bootstrap
```

Она сделает следующее:

- создаст папку resources\sass;
- добавит в папку resources\sass таблицу стилей app.scss, импортирующую:
  - необходимые шрифты — с сайта Google Fonts;
  - переменные, задающие имена используемых шрифтов и цветов, — из таблицы стилей _variables.scss;
  - остальные стили — из таблиц стилей Bootstrap;
- создаст в папке resources\sass файл _variables.scss, хранящий упомянутые ранее переменные.

  Можно изменить шрифты и цвета, используемые по умолчанию в Bootstrap, исправив значения этих переменных;
- добавит в файл веб-сценариев resources\js\bootstrap.js код, импортирующий программный код Bootstrap;
- добавит в файл package.json необходимые зависимости: Bootstrap, используемые им библиотеки jQuery, popper и транслятор языка SCSS;
- перезапишет файл webpack.mix.js, занеся в него код, указывающий компилировать таблицы стилей Bootstrap и создавать карты исходного кода.

После этого следует установить сам Bootstrap и используемые им библиотеки, набрав команду:

```
npm install
```

выполнить обработку статических файлов командой, например:

```
npm run dev
```

и привязать к веб-страницам сайта полученные в результате таблицу стилей и веб-сценарий:

```
<head>
    . . .
    <link rel="stylesheet" href="/css/app.css">
    <script src="/js/app.js"></script>
</head>
```

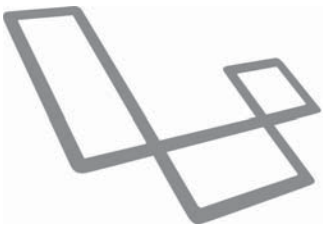
Разумеется, можно создать произвольное количество SCSS-файлов со стилями, написанными разработчиком сайта. Только тогда придется добавить в файл webpack.mix.js

выражения, предписывающие выполнить их трансляцию в CSS. Пример (предполагается, что дополнительная таблица стилей хранится в файле resources\sass\site.scss):

```
mix.js('resources/js/app.js', 'public/js')
    . . .
   .sass('resources/sass/app.scss', 'public/css/app.css')
   .sass('resources/sass/site.scss', 'public/css/site.css');
```

Получившиеся таблицы стилей app.css и site.css можно привязать к странице:

```
<head>
    . . .
    <link rel="stylesheet" href="/css/app.css">
    <link rel="stylesheet" href="/css/site.css">
    . . .
</head>
```



ГЛАВА 18

Обработка выгруженных файлов

Многие сайты позволяют пользователям выгружать на них какие-либо файлы: изображения, аудио, видео, архивы и пр. Laravel предоставляет удобные инструменты для работы с такими файлами.

18.1. Настройки подсистемы обработки выгруженных файлов

Все настройки подсистемы, обрабатывающей выгруженные файлы, хранятся в модуле config\filesystem.php:

- `disks` — ассоциативный массив *дисков* — хранилищ, на которых будут размещаться выгруженные файлы. Ключи элементов массива задают имена дисков, а значения представляют собой ассоциативные массивы с настройками соответствующих дисков. Поддерживаются следующие настройки:
  - `driver` — драйвер, обеспечивающий взаимодействие с физическим хранилищем. Изначально поддерживаются драйверы:
    - `local` — локальный диск;
    - `s3` — облачная служба Amazon S3. Для ее использования необходимо установить дополнительные библиотеки, набрав команду:

      ```
      composer require league/flysystem-aws-s3-v3 "^3.0"
      ```

    - `ftp` — сервер FTP. Для его использования необходимо установить дополнительные библиотеки, набрав команду:

      ```
      composer require league/flysystem-ftp "^3.0"
      ```

    - `sftp` — сервер SFTP. Для его использования необходимо установить дополнительные библиотеки, набрав команду:

      ```
      composer require league/flysystem-sftp-v3 "^3.0"
      ```

  - `throw` — если `true`, при неудачной попытке записать файл в хранилище будет возбуждено исключение `League\Flysystem\UnableToWriteFile`. Если `false`, исключение возбуждаться не будет, а метод, выполняющий запись, вернет в качестве результата `false`. По умолчанию: `false`.

Следующие настройки используются только драйвером `local`:

- `root` — путь к корневой папке хранилища;
- `url` — интернет-адрес «корня» диска. Указывается только для дисков, хранящих файлы, которые должны быть доступны посетителям (выводиться на страницах, загружаться по щелчкам на гиперссылках и т. п.);
- `visibility` — если `'private'`, файлы, содержащиеся в хранилище, не будут доступными посетителям (закрытые файлы), если `'public'` — будут доступны (общедоступные файлы). По умолчанию: `'private'`.

  Если дать настройке `visibility` значение `'public'`, то создаваемые в хранилище файлы получат права доступа 0664 (владелец и члены его группы имеют доступ на чтение и запись, остальные пользователи — только на чтение), а папки — права 0775 (владелец и члены его группы имеют все права, остальные — только на чтение). Если же дать настройке значение `'private'`, файлы будут получать права 0600 (владелец имеет доступ на чтение и запись, остальные не имеют доступа), а папки — права 0700 (владелец имеет все права, остальные не имеют доступа);
- `permissions` — позволяет указать другие права на доступ к файлам и папкам, создаваемым в хранилище, если права по умолчанию почему-то не подходят. Значением должен быть ассоциативный массив с элементами:
  - `file` — задает права на доступ к файлам. Значением должен быть ассоциативный массив с элементами `public` (права на доступ к общедоступным файлам) и `private` (права на доступ к закрытым файлам);
  - `dir` — задает права на доступ к папкам. Значение указывается в том же формате, что и у параметра `file`.

  Пример:

```
'local' => [
    'driver' => 'local',
    . . .
    'permissions' => [
        'file' => [
            'public' => 0664,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0775,
            'private' => 0700,
        ],
    ],
],
```

Следующие настройки используются только драйвером `s3`:

- `key` — ключ доступа пользователя (в терминологии Amazon S3 — AccessKeyId). Значение берется из локальной настройки `AWS_ACCESS_KEY_ID`, присутствующей в файле .env, но изначально «пустой»;

- secret — секретный ключ пользователя (в терминологии Amazon S3 — SecretAccessKey). Значение берется из локальной настройки AWS_SECRET_ACCESS_KEY, присутствующей в файле .env, но «пустой»;
- region — обозначение региона. Значение берется из локальной настройки AWS_DEFAULT_REGION. По умолчанию: us-east-1;
- bucket — имя используемой корзины S3. Значение берется из локальной настройки AWS_BUCKET, присутствующей в файле .env, но «пустой»;
- url — интернет-адрес «корня» облачного хранилища. Значение берется из локальной настройки AWS_URL, изначально отсутствующей в файле .env;
- endpoint — имя используемой точки контроля. Значение берется из локальной настройки AWS_ENDPOINT, изначально отсутствующей в файле .env;
- use_path_style_endpoint — специфическая настройка Amazon S3. Значение берется из локальной настройки AWS_USE_PATH_STYLE_ENDPOINT. По умолчанию: false.

Следующие настройки используются только драйверами ftp и sftp:

- host — интернет-адрес сервера;
- username — имя пользователя для подключения к серверу;
- password — пароль пользователя;
- root — путь к папке сервера, в которой будут сохраняться файлы. Чтобы сохранять файлы в «корне», следует указать у этого параметра в качестве значения «пустую» строку;
- port — номер TCP-порта, через который работает сервер, если этот номер отличается от используемого по умолчанию;
- timeout — промежуток времени, в течение которого Laravel будет пытаться подключиться к серверу, в виде числа в секундах.

Следующие настройки используются только драйвером ftp:

- passive — если true, для подключения к серверу будет использоваться пассивный режим (по умолчанию: false);
- ssl — если true, взаимодействие с сервером будет осуществляться по защищенному протоколу SSL (по умолчанию: false).

Следующие настройки используются только драйвером sftp:

- privateKey — путь к файлу закрытого ключа;
- passphrase — пароль к закрытому ключу;
- hostFingerprint — отпечаток хоста;
- maxTries — максимальное количество попыток соединения;
- userAgent — если false, вход на сервер будет выполняться средствами клиента, если true — путем прямого указания закрытого ключа (по умолчанию: false).

Изначально объявлены три хранилища:

- `local` — закрытое, хранит файлы на локальном диске в папке storage\app;
- `public` — общедоступное, хранит файлы на локальном диске в папке storage\app\public, «корень» доступен по интернет-адресу ***<адрес хоста из локальной настройки*** `APP_URL`**>/storage**.

  Изначально локальная настройка `APP_URL` хранит интернет-адрес **http://localhost**. Если сайт запускается под отладочным веб-сервером PHP, работающим через TCP-порт 8000, выгруженные файлы выводиться не будут. Чтобы они успешно выводились, следует указать в локальной настройке `APP_URL` интернет-адрес: **http://localhost:8000**;
- `s3` — облачное, сохраняет файлы в хранилище Amazon S3;

- `default` — хранилище по умолчанию, размещающее файлы на локальном диске (локальное хранилище). Берет значение из локальной настройки `FILESYSTEM_DRIVER`, изначально отсутствующей в файле .env. По умолчанию: `local`.

  Если на сайт планируется выгружать файлы, предназначенные для публикации в Сети, следует сразу же заменить локальное хранилище по умолчанию на `public`:

  ```
  'default' => env('FILESYSTEM_DRIVER', 'public'),
  ```

- `links` — перечень символических ссылок, создаваемых командой `storage:link` утилиты artisan (об этой команде речь пойдет очень скоро). Указывается в виде ассоциативного массива, ключи элементов которого должны представлять собой пути к создаваемым символическим ссылкам, а значения зададут пути, на которые указывают эти ссылки.

  Изначально содержит только ссылку `public\storage`, указывающую на путь storage\app\public.

18.2. Создание символических ссылок на папки с выгруженными файлами

Выгруженные файлы, подлежащие публикации на сайте, рекомендуется размещать в папке storage\app\public или в папках, вложенных в нее. Но веб-сервер может обслуживать лишь файлы, находящиеся в папке public или вложенных в нее папках, остальные папки проекта ему недоступны.

Выходом является создание символической ссылки, находящейся в папке public и указывающей на папку storage\app\public. Для создания таких ссылок служит команда:

```
php artisan storage:link [--relative] [--force]
```

Поддерживаются командные ключи:

- `--relative` — записать в создаваемой символической ссылке относительный путь вместо абсолютного. Это может пригодиться при переносе папки проекта по другому местоположению;
- `--force` — перезаписать существующие символические ссылки.

Команда `storage:link` создает символические ссылки, параметры которых записаны в настройке `links` модуля config\filesystem.php (см. *разд. 18.1*). Изначально там присутствует лишь ссылка `public\storage`, указывающая на папку storage\app\public. При необходимости туда можно добавить другие ссылки, например:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('avatars') => storage_path('app/other/avatars'),
],
```

18.3. Хранение выгруженных файлов

Для хранения выгруженных файлов изначально предусмотрены две папки:

- storage\app — хранит файлы, не публикуемые на сайте в изначальном виде, а предназначенные для дальнейшей обработки (например, создания на их основе пользовательских аватаров). Соответствует локальному хранилищу `local`;
- storage\app\public — хранит файлы, публикуемые в изначальном виде. Соответствует локальному хранилищу `public`.

В базе данных выгруженные файлы регистрируются в виде их путей, записанных относительно корневой папки хранилища, в которое они помещены. Для их хранения можно выделить обычное строковое поле достаточной длины. Вот фрагмент кода миграции (такое поле для примера помечено как необязательное для заполнения):

```
Schema::create('bbs', function (Blueprint $table) {
    . . .
    $table->string('pic')->nullable();
});
```

18.4. Базовые средства для обработки выгруженных файлов

У веб-формы, выгружающей файлы, следует указать метод отправки данных `multipart/form-data` (указывается в атрибуте `enctype` тега `<form>`).

18.4.1. Валидаторы для выгруженных файлов

Для проверки выгруженных файлов на соответствие заданным условиям Laravel предоставляет следующие валидаторы (в дополнение к описанным в *разд. 10.2.3*):

- `file` — любой выгруженный файл;
- `image` — файл с изображением формата: GIF, JPEG, PNG, BMP, SVG или WebP:

  ```
  $validationRules = ['pic' => 'sometimes|image];
  ```

- `mimes:<values>` — файл, имеющий одно из приведенных в списке *values* через запятую расширений (указываются без начальных точек):

  ```
  'avatar' => 'mimes:jpeg,jpg,jpe,png'
  ```

- `mimetypes:<values>` — файл, принадлежащий одному из приведенных в списке *values* через запятую MIME-типов:

  ```
  'avatar' => 'mimetypes:image/jpeg,image/png'
  ```

- `dimensions:<параметры размеров>` — файл, хранящий изображение с заданными в *параметрах* размерами. *Параметры размеров* указываются в формате *<параметр>=<значение>* и отделяются друг от друга запятыми. Поддерживаются следующие *параметры*:

  - `min_width` — минимальная ширина в пикселах;
  - `min_height` — минимальная высота в пикселах;
  - `width` — строго заданная ширина в пикселах;
  - `height` — строго заданная высота в пикселах;
  - `max_width` — максимальная ширина в пикселах;
  - `max_height` — максимальная высота в пикселах;
  - `ratio` — соотношение ширины и высоты, заданное либо в виде вещественного числа, либо в виде натуральной дроби формата *<ширина>/<высота>*.

  Пример:

  ```
  'avatar' => 'dimensions:max_width=300,max_height=200,ratio=3/2'
  ```

  Сформировать это правило также можно программно. У фасада `Illuminate\Validation\Rule` вызывается метод `dimensions()`, а у возвращенного им объекта — методы: `minWidth()`, `minHeight()`, `width()`, `height()`, `maxWidth()`, `maxHeight()` и `ratio()`. Все они в качестве результата возвращают текущий объект, что позволяет сцеплять их вызовы. Пример:

  ```
  use Illuminate\Validation\Rule;
  . . .
  'avatar' => [Rule::dimensions()->maxWidth(300)->maxHeight(200)
                                                 ->ratio(3 / 2)]
  ```

- `size:<value>` (применительно к файлам) — файл должен иметь размер *value* Кбайт;
- `min:<min>` (применительно к файлам) — файл должен иметь размер не менее *min* Кбайт;
- `max:<max>` (применительно к файлам) — файл должен иметь размер не более *max* Кбайт;
- `between:<min>,<max>` (применительно к файлам) — файл должен иметь размер от *min* до *max* Кбайт;
- `lt:<value>` (применительно к файлам) — файл должен иметь размер менее *value* Кбайт;
- `lte:<value>` (применительно к файлам) — файл должен иметь размер не более *value* Кбайт.
- `gt:<value>` (применительно к файлам) — файл должен иметь размер более *value* Кбайт;
- `gte:<value>` (применительно к файлам) — файл должен иметь размер не менее *value* Кбайт.

18.4.2. Получение выгруженных файлов

Проще всего извлечь выгруженный посетителем файл, переданный в POST-параметре, обратившись к свойству объекта запроса, чье имя совпадает с именем нужного параметра:

```
// Извлекаем файл, отправленный в POST-параметре pic
$pic = request()->pic;
```

Такое свойство хранит непосредственно сам выгруженный файл, представленный объектом класса `Illuminate\Http\UploadedFile`, который будет рассмотрен позже.

Также можно использовать следующие методы, поддерживаемые классом запроса:

- `file()` — в зависимости от формата вызова:
  - `file(<имя POST-параметра>)` — возвращает файл, полученный через POST-параметр с указанным *именем*. Если POST-параметра с таким *именем* в запросе нет, возвращает `null`. Пример:

    ```
    $pic = request()->file('pic');
    ```

  - `file()` (без параметров) — возвращает ассоциативный массив со всеми выгруженными файлами, что были получены в клиентском запросе. Ключи элементов этого массива будут иметь те же имена, что и соответствующие им POST-параметры. Пример:

    ```
    $files = request()->files();
    $pic = $files['pic'];
    $addPic = $files['add_pic'];
    ```

- `hasFile(<имя POST-параметра>)` — возвращает `true`, если в текущем клиентском запросе присутствует POST-параметр, имеющий заданное *имя* и хранящий выгруженный файл, и `false` — в противном случае:

  ```
  if (request()->hasFile('add_pic'))
      // Получена дополнительная иллюстрация
  ```

18.4.3. Получение сведений о выгруженных файлах

Класс `UploadedFile` поддерживает ряд методов, позволяющих получить различные сведения о выгруженном файле:

- `isValid()` — возвращает `true`, если текущий файл был успешно получен, и `false` — в противном случае:

  ```
  $pic = request()->file('pic');
  if ($pic->isValid())
      // Файл успешно получен
  ```

- `path()` — возвращает путь к текущему файлу;
- `extension()` — возвращает расширение текущего файла без начальной точки;
- `getClientMimeType()` — возвращает MIME-тип текущего файла;

- `getClientOriginalName()` — возвращает изначальное имя текущего файла, под которым он хранится на компьютере выгрузившего его посетителя.

Статический метод `getMaxFilesize()` возвращает максимально допустимый размер выгружаемого файла, заданный в веб-форме или настройках PHP, в виде числа в байтах.

18.4.4. Сохранение выгруженных файлов

Выгруженный файл временно сохраняется в особой папке. В процессе обработки запроса его необходимо записать на какой-либо из дисков, описанных в настройках проекта (см. *разд. 18.1*), для постоянного хранения — иначе при получении следующего запроса он будет удален.

> ***Внимание: ошибка!***
>
> В версии Laravel, описываемой в книге, присутствует ошибка, вследствие которой все рассмотренные в этом разделе методы, если при их вызове явно не указан диск, всегда сохраняют файл на диске `local`. Поэтому, если файл требуется сохранить на другом диске, его следует указать явно.
>
> Методы, описанные в остальных разделах, если диск не назначен, работают корректно — всегда обращаются к диску, указанному в настройках проекта как используемому по умолчанию.

Для сохранения выгруженных файлов применяются следующие методы класса `UploadedFile`:

- `store(<путь>[, <параметры>=[]])` — сохраняет текущий выгруженный файл по заданному *пути под автоматически сгенерированным именем* и возвращает полный путь к сохраненному файлу (этот путь можно, например, записать в поле записи базы данных). Если файл не удалось сохранить, возвращает `false`. Пример:

  ```
  $bb->pic = $pic->store('bbs');
  ```

  Если требуется сохранить файл в корневой папке диска, в качестве *пути* следует указать «пустую» строку. По умолчанию файл записывается на диск, заданный по умолчанию.

  В качестве *параметров* можно указать:

  - имя диска, на который нужно записать файл, в виде строки:

    ```
    $bb->pic = $pic->store('bbs', 'public');
    ```

  - ассоциативный массив, в котором ключи элементов задают параметры сохранения файла, а значения элементов — значения этих параметров. Поддерживаются следующие параметры:
    - `disk` — имя диска, на который записывается файл, в виде строки;
    - `visibility` — доступность файла для посетителей сайта. Значение указывается в виде значений констант `VISIBILITY_PUBLIC` (общедоступный файл) или `VISIBILITY_PRIVATE` (закрытый файл) интерфейса `Illuminate\Contracts\Filesystem\Filesystem`. Также можно использовать строки `'public'` и `'private'` соответственно.

Пример:

```
use Illuminate\Contracts\Filesystem\Filesystem;
. . .
$bb->pic = $pic->store('bbs', ['disk' => 'public',
           'visibility' => Filesystem::VISIBILITY_PUBLIC]);
```

Вместо этого метода можно использовать аналогичный метод `putFile()` фасада `Illuminate\Support\Facades\Storage`:

```
putFile(<путь>, <сохраняемый файл>[, <параметры>=[]])
```

Пример:

```
use Illuminate\Support\Facades\Storage;
. . .
$bb->pic = Storage::putFile('bbs', $pic);
```

Здесь в качестве *параметров* можно указать:

- обозначение доступности файла в виде строки или константы интерфейса `Filesystem`:

```
$bb->pic = Storage::putFile('bbs', $pic, 'private');
```

- ассоциативный массив с параметрами. Поддерживается только параметр `visibility`. Пример:

```
$bb->pic = Storage::putFile('bbs', $pic,
          ['visibility' => Filesystem::VISIBILITY_PRIVATE]);
```

Метод `putFile()` фасада `Storage` по умолчанию сохраняет файл на диске `local`. Чтобы сохранить файл на другом диске, следует воспользоваться одним из следующих методов того же фасада:

- `disk([<имя диска>=null])` — возвращает объект диска с указанным *именем*. Если *имя* не указано, возвращает объект диска `local`. Пример:

```
$user->avatar = Storage::disk('public')
                       ->putFile('other/avatars',
                                 request()->avatar);
```

- `drive([<имя диска>=null])` — то же самое, что и `disk()`.

Методы фасада `Storage` выполняются чуть быстрее аналогичных методов класса `UploadedFile`, что может быть критично при программировании высоконагруженных сайтов;

- `storePublicly(<путь>[, <параметры>=[]])` — то же самое, что и `store()`, только сразу делает сохраненный файл общедоступным:

```
$bb->pic = $pic->storePublicly('bbs', 'public');
```

- `storeAs(<путь>, <имя>[, <параметры>=[]])` — то же самое, что и `store()`, только сохраняет файл под заданным *именем*:

```
$user->avatar = request()->avatar
                       ->storeAs('other/avatars',
                                 $user->name . '_avatar.png');
```

Вместо этого метода можно использовать метод `putFileAs()` фасада `Storage`:

```
putFileAs(<путь>, <сохраняемый файл>, <имя>[, <параметры>=[]])
```

Пример:

```
$user->avatar = Storage::putFileAs('other/avatars',
                                   request()->avatar,
                                   $user->name . '_avatar.png');
```

- `storePubliclyAs(<путь>, <имя>[, <параметры>=[]])` — то же самое, что и `storeAs()`, только сразу делает сохраненный файл общедоступным.

Пример создания нового объявления и сохранения выгруженного файла, хранящего иллюстрацию к нему (файл передается через POST-параметр `pic`, записывается на диск `public`, путь к сохраненному файлу записывается в поле `pic` таблицы):

```
public function store($request) {
    $bb = new Bb($request->all());
    . . .
    $pic = $request->files('pic');
    if ($pic)
        $bb->pic = $pic->store('', 'public');
    $bb->save();
    . . .
}
```

18.4.5. Выдача выгруженных файлов посетителям

18.4.5.1. Вывод выгруженных файлов

Если выгруженный файл является изображением, аудио- или видеофайлом и при этом хранится в общедоступном хранилище, для его вывода достаточно вставить в код веб-страницы тег, соответственно `<img>`, `<audio>` или `<video>`, и поместить в него интернет-адрес нужного файла. Сформировать интернет-адрес файла с указанным *путем* позволит метод `url(<путь к файлу>)` фасада `Storage`. Пример:

```
<img src="{{ Storage::url($bb->pic) }}" alt="Иллюстрация">
```

Метод `urlTemporary()` возвращает интернет-адрес файла с заданным *путем*, который является действительным лишь до истечения указанной *временно́й отметки* (указывается в виде объекта класса `DateTime` или `Carbon`):

```
urlTemporary(<путь к файлу>, <временна́я отметка>[, <параметры>=[]])
```

Пример:

```
<img src="{{ Storage::urlTemporary($bb->pic,
                                   now()->addMinutes(1)) }}"
     alt="Эта картинка будет показываться лишь одну минуту">
```

Параметры имеет смысл указывать, если файл хранится в облаке. Они указываются в виде ассоциативного массива, ключи элементов которого задают имена параметров, а значения элементов станут значениями этих параметров. Пример:

```
<img src="{{ Storage::disk('s3')
            ->urlTemporary($bb->pic, now()->addMinutes(1),
        ['ResponseContentType' => 'application/octet-stream']) }}">
```

Если же файл хранится на закрытом диске, его можно отправить посетителю в составе отдельного ответа. Для этого достаточно вызвать метод `response()` фасада `Storage`:

```
response(<путь к файлу>[, <имя файла>=null[,
        <ассоциативный массив с добавляемыми в ответ заголовками>=[][,
        <способ обработки файла на стороне клиента>='inline']]])
```

Если *имя файла* не указано, файл будет отправлен клиенту под своим изначальным именем. В *ассоциативном массиве* ключи элементов должны соответствовать именам заголовков, а значения элементов зададут значения для этих заголовков.

Если параметру *способ обработки файла на стороне клиента* дать значение `'inline'`, фреймворк отправит в составе заголовков ответа указание веб-обозревателю непосредственно вывести этот файл на экран. Если же дать этому параметру значение `'attachment'`, веб-обозреватель получит указание сохранить этот файл на локальном диске.

Метод в качестве результата возвращает объект ответа, который следует вернуть из контроллера. Пример:

```
Route::get('/account/{user}/avatar', function (User $user) {
    return Storage::disk('local')->response($user->avatar,
                                           'avatar.jpg');
});
```

18.4.5.2. Реализация загрузки выгруженного файла

Чтобы позволить посетителям загрузить выгруженный файл, нужно создать указывающую на него гиперссылку. Если файл хранится на общедоступном диске, интернет-адрес для этой гиперссылки можно сформировать методом `url()`, описанным в *разд. 18.4.5.1*. Пример:

```
<a href="{{ Storage::url($bb->desc) }}">Загрузить описание товара</a>
```

Если же файл хранится на закрытом диске, можно пойти двумя путями:

- вызвать метод `response()` (см. *разд. 18.4.5.1*), дав последнему параметру значение: `'attachment'`:

  ```
  Route::get('/bbs/{bb}/price', function (Bb $bb) {
      return Storage::disk('local')
                    ->response($user->avatar, 'avatar.jpg', [],
                                              'attachment');
  });
  ```

- вызвать метод `download()` фасада `Storage`:

  ```
  download(<путь к файлу>[, <имя файла>=null[,
           <ассоциативный массив с добавляемыми в ответ заголовками>=[]]])
  ```

  Пример:

  ```
  return Storage::disk('local')
                ->download($user->avatar, 'avatar.jpg');
  ```

18.4.6. Удаление выгруженных файлов

Для удаления выгруженных файлов с заданными *путями* служит метод `delete()` фасада `Storage`, который поддерживает два формата вызова:

```
delete(<путь к файлу 1>, <путь к файлу 2>, ... <путь к файлу n>)
delete(<массив путей к файлам>)
```

Примеры:

```
Storage::delete($bb->pic);
Storage::disk('local')->delete($user->avatar, $user->bigAvatar);
```

Пример удаления объявления и файла, хранящего иллюстрацию к нему (путь к этому файлу хранится в поле `pic`):

```
public function destroy(Bb $bb) {
    if ($bb->pic)
        Storage::disk('public')->delete($bb->pic);
    $bb->delete();
    . . .
}
```

При правке записи, в которой хранится путь к выгруженному файлу, следует проверить, не указал ли посетитель в этой записи другой файл для сохранения, и, если это так, перед сохранением нового файла удалить старый. Вот пример правки объявления с удалением старого файла с иллюстрацией и сохранением нового (файл передается через POST-параметр `pic`):

```
public function update($request, Bb $bb) {
    $bb->fill($request->all());
    . . .
    $pic = $request->pic;
    if ($pic) {
        if ($bb->pic)
            Storage::disk('public')->delete($bb->pic);
        $bb->pic = $pic->store('', 'public');
    }
    $bb->save();
    . . .
}
```

18.5. Расширенные средства для работы с выгруженными файлами

Фасад `Storage` предоставляет ряд методов, позволяющих считывать содержимое текстовых файлов, записывать или дописывать в них строки, копировать, перемещать файлы, получать сведения о них, а также работать с папками.

Путь к файлу (папке) в вызовах этих методов указывается относительно корневой папки диска, в котором сохранен этот файл (папка).

18.5.1. Чтение из файлов и запись в них

- `put(<путь>, <содержимое>[, <параметры>=[]])` — записывает в файл с заданным *путем* указанное строковое *содержимое*. Старое содержимое файла при этом теряется. Если файл с таким *путем* еще не существует, он будет создан. *Параметры* задаются в том же формате, что и у метода `putFile()` (см. *разд. 18.4.4*).

  В качестве результата возвращается `true`, если запись в файл была успешно выполнена, и `false` — в противном случае. Если же настройке `throw` диска, на котором находится файл, дано значение `true`, в случае неуспешной записи возбуждается исключение `League\Flysystem\UnableToWriteFile` (подробности — в *разд. 18.1*). Пример:

```
>>> use Illuminate\Support\Facades\Storage;
>>> Storage::disk('local')->put('site.log',
...                            'Сайт впервые запущен!');
=> true
```

- `get(<путь>)` — возвращает содержимое текстового файла с указанным *путем* в виде строки:

```
>>> echo Storage::disk('local')->get('site.log');
Сайт впервые запущен!
```

- `prepend(<путь>, <содержимое>[, <разделитель>=PHP_EOL])` — добавляет в *начало* файла с указанным *путем* заданное строковое *содержимое*, завершая его заданным *разделителем*. Если файл с таким *путем* не существует, создает его. Возвращает тот же результат, что и метод `put()`. Пример:

```
>>> Storage::disk('local')->prepend('site.log',
...                                 '==============');
=> true
```

- `append(<путь>, <содержимое>[, <разделитель>=PHP_EOL])` — добавляет в *конец* файла с указанным *путем* заданное строковое *содержимое*, завершая его заданным *разделителем*. Если файл с таким *путем* не существует, создает его. Возвращает тот же результат, что и метод `put()`. Пример:

```
>>> Storage::disk('local')->append('site.log', '--------------');
=> true
>>> echo Storage::disk('local')->get('site.log');
==============
Сайт впервые запущен!
--------------
```

18.5.2. Получение сведений о файле

- `fileExists(<путь>)` — возвращает `true`, если файл с указанным *путем* существует, и `false` — в противном случае:

```
if (Storage::fileExists($bb->pic))
    Storage::delete($bb->pic);
```

- `directoryExists(<путь>)` — возвращает `true`, если папка с указанным *путем* существует, и `false` — в противном случае;
- `fileMissing(<путь>)` — возвращает `true`, если файл с указанным *путем*, наоборот, *не* существует, и `false` — в противном случае;
- `directoryMissing(<путь>)` — возвращает `true`, если папка с указанным *путем*, наоборот, существует, и `false` — в противном случае;
- `exists(<путь>)` — возвращает `true`, если файл или папка с указанным *путем* существует, и `false` — в противном случае:

```
if (Storage::exists($bb->pic))
    Storage::delete($bb->pic);
```

- `missing(<путь>)` — возвращает `true`, если файл или папка с указанным *путем*, наоборот, не существует, и `false` — в противном случае;
- `path(<путь>)` — возвращает полный путь к файлу с указанным *путем*;
- `size(<путь>)` — возвращает размер файла с указанным *путем* в виде числа в байтах;
- `mimeType(<путь>)` — возвращает MIME-тип файла с указанным *путем*;
- `lastModified(<путь>)` — возвращает время последнего изменения файла с указанным *путем* в виде временно́й отметки UNIX (количества секунд, прошедших с полуночи 1 января 1970 года).

18.5.3. Прочие манипуляции с файлами

- `copy(<исходный путь>, <путь назначения>)` — копирует файл с заданным *исходным путем* по *пути назначения*. Возвращает `true`, если копирование прошло удачно, и `false` — в противном случае. Пример:

```
>>> Storage::disk('local')->copy('site.log', '/logs/site.log/');
=> true
```

- `move(<исходный путь>, <путь назначения>)` — перемещает файл с заданным *исходным путем* по *пути назначения*. Возвращает `true`, если перемещение прошло удачно, и `false` — в противном случае. Пример:

```
>>> Storage::disk('local')->move('/logs/site.log',
...                             '/others/logs/main.log/');
=> true
```

- `getVisibility(<путь>)` — возвращает обозначение доступности файла с заданным *путем* посетителям сайта в виде строки `'public'` (общедоступный файл) или `'private'` (закрытый файл):

```
>>> echo Storage::disk('local')->getVisibility('site.log');
private
```

- `setVisibility(<путь>, <обозначение доступности>)` — задает для файла с указанным *путем* степень доступности для посетителей сайта, соответствующую заданному *обозначению*. Возвращает `true`, если степень доступности файла была успешно изменена, и `false` — в противном случае. Пример:

```
>>> use Illuminate\Contracts\Filesystem\Filesystem;
>>> Storage::disk('local')
...         ->setVisibility('site.log',
...                         Filesystem::VISIBILITY_PUBLIC);
=> true
```

18.5.4. Работа с папками

- `files(<путь>[, <искать файлы во вложенных папках?>=false])` — возвращает массив путей к файлам, хранящимся в папке с указанным *путем*. Если параметру *искать файлы во вложенных папках* дать значение `true`, возвращенный массив также будет содержать пути к файлам из вложенных папок. Пример:

```
>>> Storage::disk('local')->files('');
=> [ ".gitignore", "site.log" ]
>>> Storage::disk('local')->files('', true);
=> [ ".gitignore", "others/logs/main.log", "public/.gitignore",
     "site.log" ]
```

- `allFiles(<путь>)` — возвращает массив путей к файлам, хранящимся в папке с указанным *путем* и вложенных в нее папках:

```
>>> Storage::disk('local')->allFiles('');
=> [ ".gitignore", "others/logs/main.log", "public/.gitignore",
     "site.log" ]
```

- `directories(<путь>[, <искать папки во вложенных папках?>=false])` — возвращает массив путей к папкам, хранящимся в папке с указанным *путем*. Если параметру *искать папки во вложенных папках* дать значение `true`, возвращенный массив также будет содержать пути к папкам из вложенных папок. Пример:

```
>>> Storage::disk('local')->directories('');
=> [ "logs", "others", "public" ]
>>> Storage::disk('local')->directories('', true);
=> [ "logs", "others", "others/logs", "public" ]
```

- `allDirectories(<путь>)` — возвращает массив путей к папкам, хранящимся в папке с указанным *путем* и вложенных в нее папках:

```
>>> Storage::disk('local')->allDirectories('');
=> [ "logs", "others", "others/logs", "public" ]
```

- `makeDirectory(<путь>)` — создает папку с указанным *путем*. Возвращает `true`, если папка была успешно создана, и `false` — в противном случае;
- `deleteDirectory(<путь>)` — удаляет папку с указанным *путем* и все содержащиеся в ней файлы и папки. Возвращает `true`, если папка была успешно удалена, и `false` — в противном случае.

18.6. Библиотека bkwld/croppa: вывод миниатюр

Очень часто при выводе списка каких-либо позиций, содержащих графические иллюстрации, показывают не оригинальные изображения, а *миниатюры* — их уменьшенные

копии. А при переходе на страницу выбранной позиции уже демонстрируют полное изображение.

Для автоматического формирования миниатюр можно использовать библиотеку bkwld/croppa.

> ***Для успешной работы bkwld/croppa…***
> …в PHP следует активизировать расширение GD2. Без этого расширения библиотеку даже не удастся установить.

Библиотека устанавливается командой:

```
composer require bkwld/croppa
```

Далее следует открыть модуль config\app.php и:

- в список зарегистрированных в проекте провайдеров `providers` — добавить провайдер этой библиотеки:

```
'providers' => [
    . . .
    Bkwld\Croppa\CroppaServiceProvider::class,
],
```

- в список зарегистрированных обозначений фасадов `aliases` — добавить фасад этой библиотеки, дав ему обозначение `Croppa`:

```
'aliases' => Facade::defaultAliases()->merge([
    . . .
    'Croppa' => Bkwld\Croppa\Facades\Croppa::class,
])->toArray(),
```

> ***Полная документация по bkwld/croppa...***
> ...находится по адресу: **https://github.com/BKWLD/croppa**.

18.6.1. Настройки библиотеки bkwld/croppa

Настройки этой библиотеки записываются в модуле config\croppa.php. Изначально он отсутствует в папке config, и чтобы создать его, следует набрать команду:

```
php artisan vendor:publish --tag=croppa-config
```

Поддерживаются следующие настройки:

- `src_disk` — имя диска, на котором хранятся исходные изображения (по умолчанию: `public`);
- `crops_disk` — имя диска, на котором будут храниться сгенерированные миниатюры (по умолчанию: `public`).

  Автор рекомендует отвести для хранения миниатюр отдельный диск, описав его в настройках подсистемы обработки выгруженных файлов (см. *разд. 18.1*);
- `max_crops` — максимальное количество миниатюр, которые библиотека может создать для каждого из исходных изображений, в виде целого числа. Если указать зна-

чение `false`, количество создаваемых миниатюр не ограничивается. Изначально: `false`.

Эту настройку следует задать, если свободного места на диске мало и для его экономии необходимо уменьшить количество создаваемых миниатюр;

- `path` — регулярное выражение, с которым должен совпадать путь к миниатюре. В том месте регулярного выражения, в котором находится непосредственно имя файла миниатюры, должна присутствовать группа, чтобы библиотека смогла извлечь имя файла из пути. Изначально указано выражение `'storage/(.*)$'`;
- `ignore` — перечень расширений файлов, на основе которых *не* будут генерироваться миниатюры, в виде регулярного выражения. Изначально указано выражение `'\.(gif|GIF)$'` (изображения в формате GIF);
- `signing_key` — управляет генерированием электронного жетона безопасности, добавляемого к формируемым интернет-адресам миниатюр в GET-параметре `token`. Можно задать следующие значения:
  - строку с секретным ключом, подобным указываемому в рабочей настройке `app.key` (см. *разд. 3.4.2.3*), — тогда жетон будет генерироваться на основе этого ключа;
  - `'app_key'` — тогда секретный ключ для создания электронного жетона будет браться из рабочей настройки `app.key`;
  - `false` — чтобы отключить вставку в интернет-адрес жетона безопасности.

  Изначально: `'app.key'`.

  Настоятельно рекомендуется не отключать вставку электронного жетона в интернет-адрес, иначе сайт может быть подвержен атаке, при которой создается большое количество миниатюр, что вызовет переполнение дискового пространства и последующий крах сайта;
- `memory_limit` — максимальный объем оперативной памяти, отнимаемый библиотекой при генерировании миниатюр. Значение задается в том же формате, что и значение одноименной настройки PHP. Изначально: `'128M'` (128 Мбайт).

  Если приходится генерировать миниатюры на основе изображений большого объема, максимальный объем памяти следует увеличить;
- `quality` — качество создаваемых миниатюр в формате JPEG в виде целого числа от 0 (наихудшее качество) до 100 (наилучшее качество). Изначально: 95;
- `interlace` — если `true`, миниатюры JPEG будут сохраняться в формате progressive JPEG, если `false` — в sequential JPEG (изначально — `true`);
- `upsize` — если `true`, изображения, имеющие размеры меньшие, чем заданы для создания миниатюр, будут увеличиваться, что может снизить их качество. Если `false`, такие изображения увеличиваться не будут. Изначально — `false`;
- `filters` — ассоциативный массив дополнительных фильтров, которые можно наложить на генерируемые миниатюры. Ключи элементов задают обозначения этих фильтров, а значения элементов — полные пути к реализующим их классам.

Из этого массива можно удалить неиспользуемые фильтры (что незначительно уменьшит объем занимаемой памяти). Или, наоборот, в этот массив можно добавить фильтры, присутствующие в каких-либо сторонних библиотеках.

Пример задания настроек библиотеки bkwld/croppa:

```
// Модуль config\filesystems.php
return [
    . . .
    'disks' => [
        . . .
        'thumbnails' => [
            'driver' => 'local',
            'root' => storage_path('app/public/thumbnails'),
            'url' => env('APP_URL').'/storage/thumbnails',
            'visibility' => 'public',
            'throw' => false,
        ],
    ],
    . . .
];

// Модуль config\croppa.php
return [
    'src_disk' => 'public',
    'crops_disk' => 'thumbnails',
    'path' => 'storage/thumbnails/(.*)$',
    // Остальные настройки имеют значения по умолчанию
    . . .
];
```

18.6.2. Использование библиотеки bkwld/croppa

Вся функциональность библиотеки доступна через фасад `Bkwld\Croppa\Facade`, предоставляющий следующие методы:

- `url()` — генерирует и возвращает интернет-адрес миниатюры, созданной на основе исходного изображения с указанным *путем*, заданных *ширины*, *высоты* и *дополнительных настроек*:

  ```
  url(<путь к изображению>[, <ширина>=null[, <высота>=null[,
                              <дополнительные настройки>=null]]])
  ```

  Путь к изображению должен совпадать с регулярным выражением, указанным в настройке `path` (см. *разд. 18.6.1*), — в противном случае библиотека выдаст ошибку.

  Ширина и *высота* задаются в виде целых чисел в пикселах. Можно задать как оба размера, так и один из них, — в противном случае другой размер будет подобран так, чтобы пропорции изображения не исказились.

  По умолчанию библиотека сама решает, что следует сделать с исходным изображением, чтобы превратить его в миниатюру:

- если указан только один размер миниатюры — *ширина* или *высота*, — уменьшит изображение до нужных размеров:

```
<img src="{{ Croppa::url('storage/thumbnails/' . $bb->pic,
                         100) }}">
```

- если указаны оба размера — обрежет изображение с краев:

```
<img src="{{ Croppa::url('storage/thumbnails/' . $bb->pic,
                         100, 400) }}">
```

При создании миниатюры, опять же, по умолчанию будут использоваться значения настроек: `quality`, `interlace` и `upsize` (см. *разд. 18.6.1*).

Поведение библиотеки можно изменить, задавая *дополнительные настройки*. Они указываются в виде массива, элементы которого и зададут настройки миниатюры:

- `'resize'` — в процессе создания миниатюры изображение в любом случае будет уменьшаться до заданного размера, а не обрезаться (однако его пропорции могут исказиться):

```
<img src="{{ Croppa::url('storage/thumbnails/' . $bb->pic,
                         100, 40, ['resize']) }}">
```

- `'pad'[ => [<R>, <G>, <B>]]` — изображение будет уменьшено до заданных размеров с сохранением пропорций, размещено в середине создаваемой миниатюры, а незанятое пространство будет заполнено цветом с заданными составляющими *R* (красная), *G* (зеленая) и *B* (синяя). Если цвет не указан, пространство будет заполнено белым цветом. Примеры:

```
<img src="{{ Croppa::url('storage/thumbnails/' . $bb->pic,
                         100, 40, ['pad']) }}">
<img src="{{ Croppa::url('storage/thumbnails/' . $bb->pic,
                         100, 40,
                         ['pad' => [127, 127, 127]]) }}">
```

- `'quadrant'[ => <обозначение стороны>]` — для создания миниатюры будет взята часть изображения с указанными *шириной* и *высотой*, примыкающая к стороне с заданным *обозначением*. Можно указать следующие обозначения: `'T'` (верхняя сторона), `'R'` (правая), `'B'` (нижняя), `'L'` (левая) и `'C'` (середина; значение по умолчанию). Пример:

```
<img src="{{ Croppa::url('avatars/' . $user->avatar,
                         200, 200, ['quadrant' => 'T') }}">
```

- `'trim' => [<X1>, <Y1>, <X2>, <Y2>]` — миниатюра будет создана на основе прямоугольного фрагмента изображения с левым верхним углом в точке [*X1*, *Y1*] и правым нижним — в точке [*X2*, *Y2*]. Все координаты задаются относительно левого верхнего угла изображения в пикселах;

- `'trim_perc' => [<X1>, <Y1>, <X2>, <Y2>]` — то же самое, что и `trim`, только координаты задаются в процентах относительно соответствующего размера изображения;

- `'quality' => <качество>` — задает *качество* создаваемой миниатюры (перекрывает значение из настройки `quality`);

- `'interlace' => true|false` — если `true`, миниатюра будет сохранена в формате progressive JPEG, если `false` — в sequential JPEG (перекрывает значение из настройки `interlace`);
- `'upsize' => true|false` — если `true`, изображения, имеющие меньшие размеры, чем заданы для создания миниатюр, будут увеличиваться, если `false` — не будут (перекрывает значение из настройки `upsize`);
- `'filters' => <`*массив фильтров*`>` — задает фильтры, которые будут применены к создаваемой миниатюре. *Массив* должен содержать обозначения фильтров (заданные в настройке `filters`) в виде строк. Доступны следующие фильтры:
  - `gray` — черно-белое изображение;
  - `darkgray` — то же самое, что и `gray`, только изображение получается более темным;
  - `blur` — размытое изображение;
  - `negative` — негатив;
  - `orange` — изображение в оранжевых тонах;
  - `turquoise` — изображение в бирюзовых тонах.

  Примеры:

```
<img src="{{ Croppa::url('avatars/' . $user->avatar, 200, 200,
                         ['filters' => ['gray']]) }}">
<img src="{{ Croppa::url('avatars/' . $user->avatar, 200, 200,
             ['filters' => ['negative', 'turquoise']]) }}">
```

Интернет-адрес миниатюры, возвращаемый методом `url()`, имеет формат (GET-параметр `token` не показан):

<путь и изначальное имя файла>-<ширина>x<высота>-<дополнительные настройки через дефис>.<изначальное расширение файла>,

где *дополнительные настройки* записываются в формате:

<наименование настройки>[(<значения параметров настройки через запятую>)]

Например, если для изображения \storage\image.jpg запросить интернет-адрес миниатюры размерами 300×200 пикселов, созданной на основе фрагмента с левым верхним углом в точке [10%, 20%] и правым нижним углом в точке [90%, 80%], преобразованной в черно-белый вид, получится интернет-адрес:

/storage/thumbnails/image-300x200-trim_perc(10,20,90,80)-filters(gray).jpg

Миниатюры, созданные библиотекой bkwld/croppa, обрабатываются так же, как и обычные статические файлы, за одним исключением. Если выяснится, что запрашиваемый статический файл отсутствует на диске, библиотека перехватывает клиентский запрос, извлекает из него путь файла и проверяет его на совпадение с регулярным выражением из настройки `path` — т. е. выясняет, является ли запрашиваемый файл миниатюрой. Если это так, библиотека извлекает из имени запрашиваемого файла параметры миниатюры, тут же создает ее и передает запрос фреймворку на дальнейшую обработку. В результате веб-обозреватель успешно получает файл созданной «на лету» миниатюры;

- `render(<путь>)` — немедленно создает миниатюру на основе заданного *пути*. Применяется в коде контроллеров для немедленного создания миниатюр. Пример:

```
use Bkwld\Croppa\Facades\Croppa;
. . .
$thumbnail_url = Croppa::url('storage/thumbnails/' . $bb->pic,
                             100, null);
Croppa::render($thumbnail_url);
```

- `delete(<путь>)` — удаляет исходное изображение с заданным *путем* и все сгенерированные на его основе миниатюры:

```
Croppa::delete('storage/thumbnails/' . $bb->pic);
```

- `reset(<путь>)` — удаляет только миниатюры, созданные на основе исходного изображения с указанным *путем*, само исходное изображение не трогает.

18.6.3. Удаление миниатюр

Для удаления миниатюр предназначена команда, записываемая в формате:

```
php artisan croppa:purge [--filter=<шаблон>] [--dry-run]
```

По умолчанию команда удаляет все миниатюры.

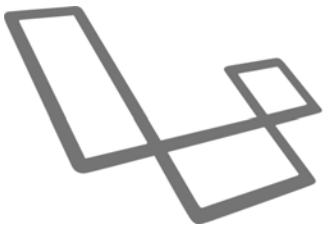
Поддерживаются командные ключи:

- `--filter` — задает *шаблон*, которому должны удовлетворять имена удаляемых миниатюр, в виде регулярного выражения. Например, команда:

```
php artisan croppa:purge --filter=.*-300x200.*
```

  удалит все миниатюры размером 300×200 пикселов;

- `--dry-run` — выводит список миниатюр, подлежащих удалению, но не удаляет их.



ГЛАВА 19

Безопасность и разграничение доступа: расширенные инструменты и дополнительная библиотека

19.1. Низкоуровневые средства для работы с пользователями

Рассмотренные в *главе 13* высокоуровневые средства для работы с пользователями (выполнения регистрации, входа, выхода, подтверждения пароля, проверки существования адреса электронной почты и сброса пароля), реализованные в готовых контроллерах, подходят в большинстве случаев. Однако для специфических применений могут оказаться полезными низкоуровневые инструменты, используемые, кстати, самими этими контроллерами.

19.1.1. Низкоуровневые средства для регистрации пользователей

Регистрация нового пользователя заключается в создании в модели `User` новой записи, содержащей сведения, которые были занесены в веб-форму регистрации. Следует помнить, что в поле пароля заносится не сам пароль, а его хеш. Пример:

```
use Illuminate\Http\Request;
use App\Models\User;
use Illuminate\Support\Facades\Hash;
. . .
public function register(Request $request) {
    $data = $request->all();
    $user = User::create(['name' => $data['name'],
                          'email' => $data['email'],
       'password' => Hash::make($data['password'])]);
    . . .
}
```

Если после успешной регистрации необходимо выполнить вход на сайт от имени только что созданного пользователя, следует воспользоваться методом `login()` или `loginUsingId()` фасада `Auth` (будут описаны в *разд. 19.1.2*).

19.1.2. Низкоуровневые средства для входа

Низкоуровневые средства для выполнения входа реализованы в виде следующих методов фасада `Illuminate\Support\Facades\Auth`:

- `attempt(<сведения о пользователе>[, <запомнить меня?>=false])` — ищет в списке пользователя с указанными *сведениями* и выполняет вход от его имени.

  Сведения о пользователе задаются в виде ассоциативного массива, ключи элементов которого должны совпадать с именами полей в таблице списка пользователей, а значения укажут величины, которые должны храниться в этих полях. Поскольку по умолчанию Laravel ищет пользователя по его адресу электронной почты и паролю, массив сведений должен включать элементы `email` и `password`, причем пароль хешировать не нужно — фреймворк сделает это сам.

  Метод возвращает `true`, если вход был выполнен успешно, и `false` — в противном случае.

  В случае успешного входа рекомендуется повторно сгенерировать серверную сессию для текущего пользователя — это позволит предотвратить ряд сетевых атак (сессии будут описаны в *разд. 26.2*). Сначала следует получить объект службы сессий, вызвав метод `session()` объекта клиентского запроса, а потом собственно перегенерировать сессию, вызвав у объекта сессии метод `regenerate()`.

  В случае удачного входа для перенаправления пользователя на страницу, на которую он пытался попасть, можно использовать метод `intended()`, поддерживаемый классом перенаправления:

```
intended([<запасной путь>='/'[, <код статуса ответа>=302[,
        <ассоциативный массив с добавляемыми в ответ ↵
         заголовками>=[][, <HTTPS?>=null]]]])
```

  Если в сессии отсутствует интернет-адрес страницы, на которую пытался попасть пользователь (такое может быть в случае, если он непосредственно зашел на страницу входа), перенаправление после успешного входа будет выполнено по заданному *запасному пути*.

  Если параметру *HTTPS* дать значение `true`, перенаправление будет выполнено по протоколу HTTPS, если дать значение `false` — по протоколу HTTP, а если `null` — по текущему протоколу.

  Пример:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
. . .
public function login(Request $request) {
    $email = $request->email;
    $password = $request->password;
    if (Auth::attempt(['email' => $email,
                       'password' => $password])) {
        $request->session()->regenerate();
        return redirect()->intended('/');
```

```
    } else {
        return back()->onlyInput('email')->withErrors(
            ['email' => 'Не найден пользователь с указанными ' .
                        'адресом и паролем']);
    }
}
```

В массиве *сведений о пользователе*, передаваемом методу `attempt()`, можно указать какие-либо дополнительные условия, которым должен удовлетворять искомый пользователь. Пример:

```
if (Auth::attempt(['email' => $email, 'password' => $password,
                   'active' => true]))
    . . .
```

Если дать параметру *запомнить меня* значение `true`, будет активизировано запоминание пользователя. Обычно этому параметру дают в качестве значения состояние флажка **Запомнить меня**, находящегося в веб-форме входа. Пример:

```
if (Auth::attempt(['email' => $email, 'password' => $password],
                  $request->boolean('remember')))
    . . .
```

- `login(<`*пользователь*`>[, <`*запомнить меня?*`>=false])` — выполняет вход от имени *пользователя*, представленного записью модели `User`. Результата не возвращает. Назначение параметра *запомнить меня* то же, что и у метода `attempt()`. Пример:

```
$editor = User::firstWhere('name', 'editor');
Auth::login($editor);
```

- `loginUsingId(<`*ключ пользователя*`>[, <`*запомнить меня?*`>=false])` — выполняет вход от имени пользователя с заданным *ключом*. Если вход увенчался успехом, возвращает запись модели `User`, хранящую пользователя с заданным *ключом*, в противном случае — `false`. Назначение параметра *запомнить меня* то же, что и у метода `attempt()`. Пример:

```
public function login($user_id) {
    if (Auth::loginUsingId($user_id) {
        return redirect()->route('home');
    } else {
        return redirect()->route('failed_login');
    }
}
```

- `viaRemember()` — возвращает `true`, если по отношению к текущему пользователю ранее было активировано запоминание, и `false`, если он выполнил обычный вход;
- `once(<`*сведения о пользователе*`>[, <`*запомнить меня?*`>=false])` — пытается выполнить временный вход от имени пользователя с указанными *сведениями*, задаваемыми в том же формате, что и у метода `attempt()`.

  Временный вход действует в течение одного клиентского запроса, после выполнения которого пользователь «выпроваживается» с сайта. Никакие cookie при этом клиенту не отсылаются, и никакие данные в серверной сессии не сохраняются.

Метод возвращает `true`, если вход был выполнен удачно, и `false` — в противном случае.

19.1.3. Низкоуровневые средства для выполнения выхода

Низкоуровневые средства для выполнения выхода реализованы в виде методов фасада `Auth`:

- `logout()` — выполняет выход с сайта.

  При вызове этого метода также генерируется новый электронный жетон запоминания пользователя. В результате попытка пользователя войти на сайт на другом устройстве (другом компьютере, смартфоне, планшете и др.), активизировав запоминание, не увенчается успехом, и ему придется выполнять вход как обычно, вводя свои адрес и пароль;

- `logoutCurrentDevice()` — выполняет выход с сайта, не перегенерируя электронный жетон запоминания пользователя.

После выхода с сайта следует выполнить два действия, чтобы предотвратить некоторые типы сетевых атак:

- пометить серверную сессию как недействительную (в результате чего она будет удалена при поступлении следующего клиентского запроса) — вызвав у объекта сессии метод `invalidate()`;
- перегенерировать и записать в новую сессию электронный жетон безопасности, используемый для защиты от межсайтовых атак, — вызовом у того же объекта метода `regenerateToken()`.

Пример:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
. . .
public function logout(Request $request) {
    Auth::logout();
    $request->session()->invalidate();
    $request->session()->regenerateToken();
    . . .
}
```

19.1.4. Низкоуровневые средства для подтверждения пароля

Для подтверждения пароля применяется метод `check()` фасада `Hash` (будет описан в *разд. 26.4.2.2*). Первым параметром этому методу передается пароль, введенный в веб-форму подтверждения, в нехешированном виде, а вторым — пароль, извлеченный из списка пользователей. Пример:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
. . .
```

```
public function check(Request $request) {
    if (Hash::check($request->password,
                    request()->user()->password)) {
        return redirect()->intended();
    } else {
        return back()->withErrors(
                        ['password' => 'Неправильный пароль']);
    }
}
```

19.1.5. Низкоуровневые средства для проверки существования адреса электронной почты

Для реализации проверки существования адреса электронной почты низкоуровневыми средствами следует выполнить действия, перечисленные далее:

1. Создать действие, которое будет выводить страницу с сообщением о том, что пользователь еще не подтвердил существование указанного им адреса.

   Маршрут, ведущий на это действие, должен иметь имя `verification.notice`. Именно на маршрут с таким именем посредник `verified` будет выполнять перенаправление, если адрес не проверен (подробности — в *разд. 13.12*).

   Пример:

   ```
   Route::get('/email/must-verified',
              [AuthController::class, 'emailMustVerified'])
        ->middleware('auth')->name('verification.notice');
   . . .
   class AuthController extends Controller {
       . . .
       public function emailMustVerified() {
           return view('auth.email-must-verified');
       }
   }
   ```

2. Создать действие, повторно отправляющее электронное письмо с просьбой подтвердить существование адреса. Это необходимо предусмотреть на тот случай, если ранее отправленное письмо не дойдет до пользователя. Действие будет выполняться после нажатия кнопки отправки данных в веб-форме проверки адреса (как правило, эта веб-форма содержит только кнопку отправки данных).

   В теле действия следует выполнить отправку письма вызовом у объекта текущего пользователя метода `sendEmailVerificationNotification()`.

   Пример:

   ```
   Route::post('/email/send-notification',
               [AuthController::class, 'sendNotification'])
        ->middleware(['auth', 'throttle:6,1'])
        ->name('verification.send');
   . . .
   use Illuminate\Http\Request;
   ```

```
class AuthController extends Controller {
    . . .
    public function sendNotification(Request $request) {
        $request->user()->sendEmailVerificationNotification();
        return view('auth.notification-sent');
    }
}
```

3. Создать действие, помечающее адрес электронной почты как проверенный. Оно будет выполняться при щелчке на гиперссылке подтверждения существования адреса, отправленного в электронном письме.

   Маршрут, ведущий на это действие, должен содержать шаблонный путь **/email/verify/{id}/{hash}**. С URL-параметром `id` передается ключ пользователя, а с URL-параметром `hash` — электронная подпись интернет-адреса. С маршрутом должен быть связан посредник `signed`, требующий, чтобы интернет-адрес включал электронную подпись.

   Само действие в качестве параметра должно принимать объект формального запроса `Illuminate\Foundation\Auth\EmailVerificationRequest`. Пометка адреса электронной почты как проверенного выполняется вызовом у этого объекта метода `fulfill()`.

   Пример:

```
Route::get('/email/verify/{id}/{hash}',
           [AuthController::class, 'emailVerify'])
     ->middleware(['auth', 'signed'])
     ->name('verification.verify');
. . .
use Illuminate\Foundation\Auth\EmailVerificationRequest;
class AuthController extends Controller {
    . . .
    public function emailVerify(EmailVerificationRequest
                                $request) {
        $request->fulfill();
        return view('auth.email-verified');
    }
}
```

19.1.6. Низкоуровневые средства для сброса пароля

Для реализации сброса пароля низкоуровневыми средствами следует выполнить действия, перечисленные далее:

1. Создать действие, которое будет выводить страницу с веб-формой сброса пароля, содержащей поле для ввода адреса электронной почты. По этому адресу будет отправлено письмо с гиперссылкой для сброса пароля. Пример:

```
Route::get('/forgot-password',
           [AuthController::class, 'passwordRequest'])
     ->middleware('guest')->name('password.request');
. . .
```

```
class AuthController extends Controller {
    . . .
    public function passwordRequest() {
        return view('auth.password-request');
    }
}
```

2. Создать действие, отправляющее электронное письмо с гиперссылкой для сброса пароля. Оно будет выполняться после нажатия кнопки отправки данных в веб-форме ввода адреса электронной почты.

   В теле этого действия следует произвести отправку письма вызовом метода `sendResetLink()` у фасада `Illuminate\Support\Facades\Password`:

```
sendResetLink(<сведения о пользователе>[,
              <анонимная функция>=null])
```

   Сведения о пользователе передаются в виде ассоциативного массива, ключи элементов которого должны соответствовать полям таблицы, в которой хранится список пользователей, а значения элементов зададут значения, по которым будет производиться поиск пользователя. Как правило, в составе *сведений о пользователе* передается только его адрес электронной почты (`email`).

   Если *анонимная функция* не указана, будет выполнена отправка электронного письма вызовом у объекта найденного пользователя метода `sendPasswordResetNotification()` (описан в *разд. 13.13.1*). В противном случае будет вызвана указанная *анонимная функция*, и ей в качестве параметров будут переданы объект найденного пользователя и электронный жетон, помещаемый в состав интернет-адреса гиперссылки сброса пароля. В теле этой *функции* можно произвести отправку электронного письма самостоятельно.

   Метод `sendResetLink()` вернет строковое обозначение статуса отправки письма. Чтобы выяснить, было ли письмо отправлено, следует сравнить его со значениями следующих констант фасада `Password`:

   - `RESET_LINK_SENT` — письмо было успешно отправлено;
   - `INVALID_USER` — пользователь с указанным адресом электронной почты не существует в списке;
   - `RESET_THROTTLED` — слишком много запросов на отправку письма.

   Пример:

```
Route::post('/forgot-password',
            [AuthController::class, 'passwordEmail'])
     ->middleware('guest')->name('password.email');
. . .
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;
class AuthController extends Controller {
    . . .
    public function passwordEmail(Request $request) {
        $request->validate(['email' => 'required|email']);
        $status = Password::sendResetLink([$request->email]);
```

```
        if ($status === Password::RESET_LINK_SENT) {
            return view('auth.password-email-sent');
        } else {
            return back()
                ->withErrors(['email' => 'Неправильный адрес']);
        }
    }
}
```

3. Создать действие, которое будет выводить страницу с веб-формой для собственно сброса пароля. Она будет вызываться при щелчке на гиперссылке сброса, отправленной в составе электронного письма.

   Маршрут, ведущий на это действие, должен содержать шаблонный путь **/reset-password/{hash}**. С URL-параметром `hash` передается электронный жетон для сброса пароля. Этот жетон следует поместить в состав веб-формы сброса пароля, выводимой на странице, в скрытом поле.

   Пример:

```
Route::get('/reset-password/{token}',
           [AuthController::class, 'passwordReset'])
    ->middleware('guest')->name('password.reset');
. . .
class AuthController extends Controller {
    . . .
    public function passwordReset($token) {
        return view('auth.password-request',
                    ['token' => $token]);
    }
}
```

4. Создать действие, которое будет сохранять пароль, введенный в веб-форму. Оно будет вызываться при нажатии кнопки отправки данных в веб-форме сброса пароля.

   В теле этого действия необходимо вызвать метод `reset()` фасада `Password`:

   `reset(<`*сведения о пользователе*`>, <`*анонимная функция*`>)`

   Сведения о пользователе задаются в том же формате, что и у метода `sendResetLink()`. В их составе следует передать адрес электронной почты (`email`), новый пароль (`password`), подтверждение нового пароля (`password_confirmation`) и электронный жетон (`token`).

   Как только соответствующий пользователь будет найден, вызовется указанная *анонимная функция*. В качестве параметров она получит объект текущего пользователя и новый пароль.

   В теле этой *функции* следует, прежде всего, записать в состав сведений о пользователе хеш нового пароля. Для этого можно использовать метод `fill()` (см. *разд. 6.1.2*) или аналогичный ему `forceFill()`, способный заносить значения во все поля, а не только в помеченные как доступные для массового присваивания.

   Далее следует создать новый электронный жетон запоминания пользователя. Для этого применяется метод `setRememberToken(<`*жетон*`>)`, поддержка которого добавляет-

ся модели `User` трейтом `Authenticable`. Жетон указывается в виде строки, состоящей из 60 случайно выбранных символов (для генерирования ее можно использовать строковый метод `random()` (см. *разд. 14.1.7*).

Метод `reset()` вернет строковое обозначение статуса сохранения. Чтобы выяснить, был ли пароль сохранен, следует сравнить его со значениями следующих констант фасада `Password`:

- `PASSWORD_RESET` — пароль успешно сохранен;
- `INVALID_TOKEN` — неправильный электронный жетон.

Пример

```
Route::post('/reset-password',
            [AuthController::class, 'passwordUpdate'])
     ->middleware('guest')->name('password.update');
. . .
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;
class AuthController extends Controller {
    . . .
    public function passwordUpdate(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, $password) {
                $user->forceFill(['password' =>
                                  Hash::make($password)]);
                $user->setRememberToken(Str::random(60));
                $user->save();
            }
        );
        if ($status === Password::PASSWORD_RESET) {
            return view('auth.password-updated');
        } else {
            return back()
                 ->withErrors(['email' => 'Неправильный адрес']);
        }
    }
}
```

19.1.7. Корректная правка пароля

После правки пароля может оказаться полезным принудительно завершить пользовательские сессии, открытые на других устройствах, не завершая сессию на текущем уст-

ройстве. После этого тот же пользователь должен будет повторно войти на сайт на этих устройствах, введя свои адрес электронной почты и новый пароль. Это делается для предотвращения возможных неполадок, связанных с изменением пароля.

Прежде всего, необходимо привязать ко всем маршрутам, доступным только зарегистрированным пользователям после выполнения входа, посредник `auth.session`:

```
Route::middleware(['auth', 'auth.session'])->group(function () {
    // Маршруты
});
```

Сохранение нового пароля текущего пользователя и принудительное завершение его пользовательских сессий на других устройствах выполняет метод `logoutOtherDevices()` фасада `Auth`:

```
logoutOtherDevices(<новый пароль>[,
                   <имя поля таблицы, хранящего пароль>='password'])
```

Пример:

```
public function passwordUpdate(Request $request) {
    Auth::logoutOtherDevices($request->password);
    return redirect()->route('home');
}
```

19.2. Библиотека Laravel Socialite: вход через сторонние интернет-службы

В настоящее время очень многие пользователи Интернета являются подписчиками какой-либо интернет-службы — например, социальной сети (а то и не одной). Неудивительно, что появились решения, позволяющие выполнять регистрацию в списках пользователей различных сайтов и вход на них посредством сторонних интернет-служб. Одно из таких решений — дополнительная библиотека Laravel Socialite. Она позволяет выполнять вход посредством нескольких десятков интернет-служб, включая «ВКонтакте», Facebook, Twitter, GitHub, Instagram и др.

В этом разделе описана установка и использование библиотеки для выполнения регистрации и входа на сайт посредством социальной сети «ВКонтакте».

> ***Полная документация по Laravel Socialite...***
> ...находится по адресу: **https://laravel.com/docs/9.x/socialite**, а инструкции по использованию ее провайдеров, обеспечивающих взаимодействие с различными интернет-службами, — по адресу: **https://socialiteproviders.com/**.

19.2.1. Создание приложения «ВКонтакте»

Чтобы успешно выполнять регистрацию и вход на сайт посредством «ВКонтакте», необходимо предварительно зарегистрировать в этой социальной сети новое приложение. Вот шаги, которые необходимо выполнить для этого:

1. Выполните вход в социальную сеть «ВКонтакте». Если вы не являетесь подписчиком этой сети, предварительно зарегистрируйтесь в ней.

2. Перейдите по интернет-адресу: **https://vk.com/apps?act=manage** — чтобы попасть на страницу списка приложений, созданных текущим пользователем.
3. Нажмите кнопку **Создать** — чтобы перейти на страницу создания нового приложения.
4. Заполните форму со сведениями о создаваемом приложении (рис. 19.1). Название приложения, заносимое в одноименное поле ввода, может быть произвольным. В группе **Платформа** следует выбрать переключатель **Сайт**. В поле ввода **Адрес сайта** заносится интернет-адрес сайта, на который будет выполняться вход через «ВКонтакте», а в поле ввода **Базовый домен** — его домен. Если сайт в текущий момент еще разрабатывается и развернут на локальном хосте, следует ввести интернет-адрес: **http://localhost** и домен **localhost** соответственно. Введя все нужные данные, нажмите кнопку **Подключить сайт**.

   Возможно, придется запросить SMS-подтверждение на создание нового приложения.
5. Запросите SMS-предупреждение, следуя появляющимся на экране инструкциям.

   Откроется страница, содержащая полные сведения о создаваемом приложении.
6. Щелкните на гиперссылке **Настройки**.

   Появится веб-форма с настройками приложения.
7. Найдите поля **ID приложения** и **Защищенный ключ** (рис. 19.2). Защищенный ключ изначально скрыт.
8. Щелкните на значке глаза, расположенном в правой части поля **Защищенный ключ**, и запросите еще одно SMS-подтверждение, чтобы просмотреть этот ключ.
9. Перепишите куда-нибудь ID приложения и его ключ — они вам скоро пригодятся.

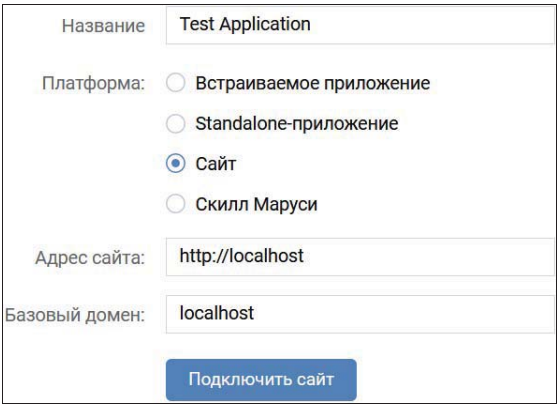


Рис. 19.1. Веб-форма для создания нового приложения «ВКонтакте»

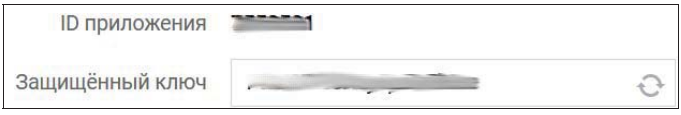


Рис. 19.2. Поля **ID приложения** и **Защищенный ключ** в веб-форме настроек приложения «ВКонтакте»

Храните в тайне сведения о приложении «ВКонтакте»!
В особенности это касается защищенного ключа.

19.2.2. Установка и настройка Laravel Socialite

Для установки самой библиотеки нужно набрать команду:

```
composer require laravel/socialite
```

Провайдер, обеспечивающий работу с «ВКонтакте», устанавливается командой:

```
composer require socialiteproviders/vkontakte
```

После установки надо выполнить следующие действия:

- открыть модуль config\app.php и:
  - в список зарегистрированных в проекте провайдеров `providers` — добавить провайдер библиотеки:

```
'providers' => [
    . . .
    SocialiteProviders\Manager\ServiceProvider::class,
],
```

  - в список зарегистрированных обозначений фасадов `aliases` — добавить фасад `Laravel\Socialite\Facades\Socialite`, дав ему одноименное обозначение:

```
'aliases' => Facade::defaultAliases()->merge([
    . . .
    'Socialite' =>
                Laravel\Socialite\Facades\Socialite::class,
])->toArray(),
```

- в массив из свойства `listen` провайдера `App\Providers\EventServiceProvider` добавить событие `SocialiteProviders\Manager\SocialiteWasCalled` и его обработчик:

```
class EventServiceProvider extends ServiceProvider {
    protected $listen = [
        . . .
        \SocialiteProviders\Manager\SocialiteWasCalled::class =>
        [
            \SocialiteProviders\VKontakte\↳
            VKontakteExtendSocialite::class,
        ],
    ];

    . . .
}
```

  События и их обработка будут описаны в *главе 22*;

- открыть модуль config\services.php, в котором записываются параметры подключения к сторонним интернет-службам, и добавить туда настройку `vkontakte`. Значением этой настройки должен быть ассоциативный массив со следующими элементами:

- `client_id` — ID приложения «ВКонтакте»;
- `client_secret` — защищенный ключ приложения;
- `redirect` — полный интернет-адрес, по которому «ВКонтакте» обратится, чтобы переслать сайту сведения о найденном пользователе.

Пример:

```
return [
    . . .
    'vkontakte' => [
        'client_id' => 'XXXXXXXXX',
        'client_secret' => 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
        'redirect' => 'http://localhost:8000/login/vk/callback'
    ],
];
```

Впрочем, в ряде случаев удобнее вынести значения этих настроек в файл .env (то есть в локальные настройки), а в модуле config\services.php написать код, извлекающий эти значения и присваивающий их рабочим настройкам:

```
// Файл .env
VKONTAKTE_KEY=XXXXXXXX
VKONTAKTE_SECRET=XXXXXXXXXXXXXXXXXXXXXXXXXXXX
VKONTAKTE_REDIRECT_URI=login/vk/callback

// Модуль config\services.php
return [
    . . .
    'vkontakte' => [
        'client_id' => env('VKONTAKTE_KEY'),
        'client_secret' => env('VKONTAKTE_SECRET'),
        'redirect' => env('APP_URL') . '/' .
                      env('VKONTAKTE_REDIRECT_URI')
    ],
];
```

Здесь полный интернет-адрес, заносящийся в настройку `services.vkontakte.redirect`, составляется из значений локальных настроек `APP_URL` и `VKONTAKTE_REDIRECT_URI`.

19.2.3. Использование Laravel Socialite

Чтобы реализовать вход через сторонние интернет-службы, в каком-либо контроллере следует объявить два действия:

- вызывающееся, когда посетитель пытается выполнить вход через стороннюю интернет-службу. Это действие запросит у сторонней интернет-службы сведения о текущем пользователе и укажет передать их по интернет-адресу из рабочей настройки `services.vkontakte.redirect`;
- вызывающееся, когда сторонняя интернет-служба обращается по полученному от первого действия интернет-адресу. Это действие получит от интернет-службы све-

дения о текущем пользователе, найдет в списке этого пользователя и выполнит вход от его имени. Если такого пользователя в списке нет, действие создаст его и, опять же, выполнит вход от его имени.

Удобнее всего объявить эти действия в контроллере `Auth\LoginController` — тогда логика, выполняющая вход на сайт, будет находиться в одном месте.

19.2.3.1. Действие первое: обращение к сторонней интернет-службе

В первом действии сначала необходимо получить объект, представляющий провайдера, который будет взаимодействовать с интернет-службой. Получить его можно, вызвав у фасада `Socialite` метод `with(<обозначение провайдера>)`. Для обращения к службе «ВКонтакте» следует указать в качестве *обозначения провайдера* строку `'vkontakte'`.

Вместо метода `with()` можно использовать метод `driver()`, имеющий тот же формат вызова.

У полученного объекта провайдера нужно вызвать метод `redirect()`, возвращающий объект перенаправления на интернет-службу. Этот объект следует вернуть из действия в качестве результата.

Пример кода первого действия:

```
use Laravel\Socialite\Facades\Socialite;
class LoginController extends Controller {
    . . .
    public function redirectToProvider() {
        return Socialite::with('vkontakte')->redirect();
    }
    . . .
}
```

19.2.3.2. Действие второе: поиск (регистрация) пользователя и вход

Сначала следует получить объект, представляющий текущего пользователя сторонней интернет-службы. Это выполняется вызовом метода `user()` объекта провайдера (как его получить, было показано в *разд. 19.2.3.1*).

Объект пользователя интернет-службы поддерживает следующие методы, дающие возможность узнать сведения о нем:

- `getId()` — возвращает уникальный ключ пользователя, под которым он зарегистрирован на сторонней интернет-службе, в виде строки;
- `getNickname()` — возвращает регистрационное имя (логин) пользователя, под которым он зарегистрирован на сторонней интернет-службе;
- `getName()` — возвращает настоящее имя пользователя;
- `getEmail()` — возвращает адрес электронной почты;
- `getAvatar()` — возвращает полный интернет-адрес файла с пользовательским аватаром.

Получив адрес электронной почты, можно найти по нему зарегистрированного на сайте пользователя. Если такой адрес электронной почты в списке пользователей отсутствует, необходимо создать нового пользователя, указав у него по крайней мере регистрационное имя и адрес электронной почты. В качестве пароля у нового пользователя задается хеш случайной строки, которую можно сгенерировать вызовом статического метода `random()` класса `Str` (см. *разд. 14.1.7*).

Все сторонние интернет-службы проверяют существование адресов электронной почты регистрирующихся на них пользователей, так что проводить эту проверку повторно нет смысла. Поэтому вновь зарегистрированного на сайте пользователя следует пометить как прошедшего проверку адреса. Это можно сделать, вызвав у него метод `markEmailAsVerified()`, который, помимо всего прочего, еще и сохранит пользователя.

Вот пример кода второго действия:

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Str;
use Illuminate\Support\Facades\Hash;
use Laravel\Socialite\Facades\Socialite;
use App\Models\User;
class LoginController extends Controller {
    . . .
    public function handleProviderCallback() {
        $sUser = Socialite::with('vkontakte')->user();
        if (!$user = User::firstWhere('email', $sUser->getEmail())) {
            $user = new User(['name' => $sUser->getNickname(),
                              'email' => $sUser->getEmail(),
                        'password' => Hash::make(Str::random(10))]);
            $user->markEmailAsVerified();
        }
        Auth::login($user);
        return redirect($this->redirectTo);
    }
}
```

После входа выполняется перенаправление по пути, записанном в свойстве `redirectTo` контроллера (показано в *разд. 13.5*).

19.2.3.3. Завершающие операции: создание маршрутов и гиперссылки входа

Осталось только добавить маршруты, ведущие на эти действия, в модуль routes\web.php:

```
use App\Http\Controllers\Auth\LoginController;
Route::get('login/vk',
           [LoginController::class, 'redirectToProvider'])
     ->name('login.vk');
Route::get('login/vk/callback',
           [LoginController::class, 'handleProviderCallback']);
```

В маршруте, ведущем на второе действие, шаблонный путь должен совпадать с путем из интернет-адреса, записанного в рабочей настройке `services.vkontakte.redirect`.

И можно создавать гиперссылку входа через стороннюю интернет-службу:

```
<a href="{{ route('login.vk') }}">Войти через ВКонтакте</a>
```

При щелчке на такой гиперссылке будет выполнено перенаправление на стороннюю интернет-службу. Последняя, если пользователь еще не вошел на нее, попросит выполнить вход и, возможно, запросит разрешение на доступ к сведениям о текущем пользователе. После чего будет выполнено перенаправление назад, на сайт, с одновременным входом на него.

19.3. Защита от атак CSRF

При атаке CSRF (Cross Sire Request Forgery, межсайтовая подделка запроса) данные, введенные на сайте, принадлежащем злоумышленнику, тайно перенаправляются на легальный сайт (например, сайт платежной системы) для выполнения нелегальной операции.

Для защиты от такого рода атак Laravel при создании каждой страницы генерирует особый электронный жетон. Одна его копия помещается в серверную сессию, а другая вставляется в скрытое поле `_token`, помещаемое в веб-форме (это скрытое поле создается директивой шаблонизатора `@csrf`). При получении данных из веб-формы фреймворк сравнивает электронный жетон, взятый из данных этой веб-формы, с его копией, хранившейся в сессии. Если обе копии идентичны, значит, данные были отправлены с того же сайта и им можно доверять, в противном случае выдается ошибка 419 (страница устарела).

Проверку жетонов выполняет посредник `App\Http\Middleware\VerifyCsrfToken`, который изначально связан со всеми веб-маршрутами.

В защищенном свойстве `except` его класса можно указать массив путей или интернет-адресов, в отношении которых не следует проверять электронные жетоны. В путях и интернет-адресах допускается использовать литерал `*`, обозначающий произвольное количество любых символов. Пример:

```
class VerifyCsrfToken extends Middleware {
    protected $except = ['stripe/*', 'https://www.trustedsite.ru/'];
}
```

Если на странице присутствуют веб-сценарии, отправляющие сайту какие-либо данные из веб-формы, в состав этих данных также должен быть включен электронный жетон безопасности. Вставить этот жетон в код страницы можно, поместив его в метатег `csrf-token` и использовав для вставки самого жетона функцию-хелпер `csrf_token()`. Пример:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Веб-сценарий может извлечь этот жетон и поместить в заголовок `X-CSRF-TOKEN` отправляемого сайту запроса, чтобы Laravel смог сверить жетон с сохраненной в сессии копией.

В шаблоне layouts\app.blade.php, сгенерированном командой `ui:auth` утилиты artisan, уже присутствует приведенный в примере метатег.

Помимо того, Laravel отправляет клиенту cookie под именем `XSRF-TOKEN`, хранящий тот же электронный жетон. Веб-сценарий может извлечь этот жетон и отправить в заголов-

ке X-XSRF-TOKEN запроса сайту, чтобы Laravel смог проверить его на идентичность сохраненному в сессии. Некоторые JavaScript-библиотеки, в частности Angular и axios, сами извлекают и отправляют этот жетон.

Если cookie XSRF-TOKEN не нужен, его создание можно отключить, переопределив в классе посредника защищенное свойство addHttpCookie и дав ему значение false:

```
class VerifyCsrfToken extends Middleware {
    protected $addHttpCookie = false;
    . . .
}
```

19.4. Ограничители частоты запросов

Сайты часто ограничивают *частоту запросов*, т. е. количество запросов, обрабатываемых в течение определенного промежутка времени (обычно одной минуты). Для этого применяются специальные инструменты, называемые *ограничителями частоты запросов*.

Если количество запросов, допустимое в течение определенного времени, исчерпано, такой ограничитель отправит в составе ответа предложение выждать указанное время перед отправкой очередного запроса. Если клиент все-таки отправит новый запрос до истечения указанного времени, то получит ошибку с кодом 429 (слишком много запросов).

> ***Ограничители частоты запросов…***
>
> …используют кеш проекта для хранения нужных для работы данных (в частности, количества запросов, уже сделанных в течение указанного времени). О подсистеме кеширования будет рассказано в *главе 29*.

19.4.1. Простейшие ограничители частоты запросов

Наиболее простая разновидность ограничителей позволяет лишь задать количество допустимых запросов в минуту и промежуток времени, в течение которого клиенту следует выждать, в случае превышения допустимого количества.

Простейший ограничитель частоты запросов реализуется посредником Illuminate\Routing\Middleware\ThrottleRequests (короткое обозначение — throttle). Обозначение этого посредника вместе с параметрами записывается в формате:

```
throttle[:<допустимое количество запросов в минуту>=60↵
[,<время ожидания в минутах>=1]]
```

Примеры:

```
// Параметры по умолчанию
Route::get('rubrics', [BbController::class, 'index'])
     ->middleware('throttle');
// Допускается не более 30 запросов в минуту,
// время ожидания по умолчанию
Route::get('rubrics', [BbController::class, 'index'])
     ->middleware('throttle:30');
```

```
// Допускается не более 30 запросов в минуту,
// время ожидания 2 минуты
Route::get('rubrics', [BbController::class, 'index'])
     ->middleware('throttle:30,2');
```

Если кеширование в проекте реализовано на основе базы данных Redis, вместо посредника `ThrottleRequest` можно использовать посредник `Illuminate\Routing\Middleware\ThrottleRequestsWithRedis`, специально оптимизированный под такой случай. Для этого достаточно в массиве из свойства `routeMiddleware` корневого класса `App\Http\Kernel` найти элемент с ключом `throttle` и записать в качестве его значения полный путь к классу упомянутого ранее посредника. Вот так:

```
class Kernel extends HttpKernel {
    . . .
    protected $routeMiddleware = [
        . . .
        'throttle' =>
            \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
        . . .
    ];
}
```

19.4.2. Именованные ограничители частоты запросов

Именованный ограничитель частоты запросов предоставляет более гибкие средства для указания допустимой частоты запросов. Он может задавать разные значения частоты для разных пользователей, разных интернет-адресов, по которым выполняется запрос, и др. Такой ограничитель имеет уникальное имя, используемое для связывания его с маршрутами.

Изначально в проекте создан именованный ограничитель `api`, задающий для всех запросов частоту 60 запросов в минуту, время ожидания в одну минуту и связанный со всеми API-маршрутами.

Код, создающий именованные ограничители частоты запросов, записывается в теле защищенного метода `configureRateLimiting()` провайдера `App\Providers\RouteServiceProvider`. Вызов этого метода присутствует в теле метода `boot()` того же провайдера, так что метод `configureRateLimiting()` исполняется при инициализации сайта.

Именованный ограничитель частоты запросов с заданным *именем* создается вызовом метода `for()` фасада `Illuminate\Support\Facades\RateLimiter`:

```
for(<имя ограничителя>, <анонимная функция>)
```

Заданная *анонимная функция* должна принимать в качестве параметра объект текущего клиентского запроса и возвращать объект, представляющий созданный ограничитель.

Для создания объекта ограничителя применяются следующие статические методы класса `Illuminate\Cache\RateLimiting\Limit`:

- `perMinute(<количество запросов>)` — создает и возвращает ограничитель частоты запросов с указанным допустимым *количеством запросов* в минуту (время ожидания всегда равно 1 минуте):

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;
. . .
class RouteServiceProvider extends ServiceProvider {
    . . .
    protected function configureRateLimiting() {
        RateLimiter::for('api', function (Request $request) {
            return Limit::perMinute(60);
        });
    }
}
```

- `perMinutes(<`*количество запросов*`>, <`*время ожидания*`>)` — создает и возвращает ограничитель частоты запросов с указанными допустимым *количеством запросов* в минуту и *временем ожидания* в минутах;
- `perHour()` — создает и возвращает ограничитель с указанными допустимым *количеством запросов* в час и *временем ожидания*, заданным в часах:

  `perHour(<`*количество запросов*`>[, <`*время ожидания*`>=1])`

  Пример:

  ```
  RateLimiter::for('rare', function (Request $request) {
      return Limit::perHout(10, 2);
  });
  ```

- `perDay()` — создает и возвращает ограничитель с указанными допустимым *количеством запросов* в сутки и *временем ожидания*, заданным в сутках. Формат вызова такой же, как и у метода `perHour()`;
- `none()` — создает и возвращает ограничитель, задающий неограниченную частоту запросов.

  Пример указания у администраторов неограниченной частоты запросов, а у остальных пользователей — частоты 60 запросов в минуту:

  ```
  RateLimiter::for('rare', function (Request $request) {
      if ($request->user()->role == 'admin') {
          return Limit::none();
      } else {
          return Limit::perMinute(60);
      }
  });
  ```

У объекта ограничителя, возвращенного этими методами, можно вызвать два следующих метода:

- `response(<`*анонимная функция*`>)` — задает ответ, возвращаемый посетителю вместо стандартного. Ответ должен генерироваться в заданной *анонимной функции*, принимающей в качестве параметров объект текущего запроса и ассоциативный массив заголовков, помещаемых в ответ. *Функция* должна создавать ответ и возвращать его в качестве результата. Пример:

```
RateLimiter::for('api', function (Request $request) {
    return Limit::perMinute(60)
         ->response(function (Request $request, array $headers) {
               return response('Помедленнее! ' .
                               'Не успеваю за вами!', 429,
                               $headers);
           });
});
```

По умолчанию создаваемый ограничитель частоты запросов действует на все запросы, приходящие на сайт, вне зависимости от IP-адреса, с которого они поступили, содержащихся в них GET- и POST-параметров и пр.;

- `by(<ключевое значение>)` — предписывает текущему ограничителю действовать лишь на те запросы, что содержат указанное *ключевое значение*.

  Пример указания частоты запросов для отдельного клиента, идентифицируемого по его IP-адресу:

```
RateLimiter::for('api', function (Request $request) {
    return Limit::perMinute(60)->by($request->ip());
});
```

  Пример создания ограничителя, разрешающего править одно и то же объявление не чаще одного раза в минуту:

```
RateLimiter::for('bb_edit', function (Request $request) {
    return Limit::perMinute(1)->by($request->input('title'));
});
```

Именованные ограничители частоты запросов могут содержать массивы из произвольного количества отдельных ограничений. В таком случае ограничения будут действовать одновременно и проверяться в том порядке, в котором присутствуют в созданном массиве. Пример создания ограничителя частоты запросов с двумя ограничениями: первое разрешает выполнять максимум 60 запросов в минуту, а второе — править одно и то же объявление не чаще одного раза в сутки:

```
RateLimiter::for('bb', function (Request $request) {
    return [
        Limit::perMinute(60),
        Limit::perDay(1)->by($request->input('title'))
    ];
});
```

Для указания созданного ограничителя частоты запросов у посредника `throttle` применяется синтаксис `throttle:<имя ограничителя>`. Пример:

```
Route::get('bbs', [BbController::class, 'index'])
     ->middleware('throttle:bb');
```

19.4.3. Низкоуровневые ограничители частоты запросов

Низкоуровневые ограничители применяются в случаях, когда необходимо ограничивать частоту запросов по-разному: в зависимости от каких-либо условий или реагиро-

вать на превышение частоты запросов не отправкой сообщения с кодом статуса 429, а как-либо иначе. Они реализуются в виде следующих методов фасада `RateLimiter`:

- `attempt()` — если указанное *количество запросов* в минуту еще не превышено, выполняет заданную *анонимную функцию*, не принимающую параметров. Если *функция* выдает результат, возвращает его, в противном случае — значение `true`. Если *количество запросов* превышено, и заданное *время ожидания* в секундах не истекло, просто возвращает `false`. Формат вызова:

  ```
  attempt(<имя ограничителя>, <количество запросов>,
          <анонимная функция>[, <время ожидания>=60])
  ```

  Программная логика, к которой применяется ограничение частоты запросов, реализуется в *анонимной функции*.

  Задаваемое в виде строки *имя* низкоуровневого *ограничителя* должно быть уникальным. Оно служит для сохранения в кеше данных, используемых созданным ограничителем (в частности, количества уже выполненных запросов).

  Пример:

  ```
  use Illuminate\Support\Facades\RateLimiter;
  class BbController extends Controller {
      public function index() {
          $result = RateLimiter::attempt('bb-list',60,function () {
                        $context = ['bbs' => Bb::latest()->get()];
                        return view('index', $context);
                    }, 30);
          if ($result) {
              return $result;
          } else {
              return response('Слишком много запросов', 429);
          }
      }
  }
  ```

- `tooManyAttempts()` — возвращает `true`, если у низкоуровневого ограничителя с заданным *именем* превышено указанное *количество запросов*, и `false` — в противном случае:

  ```
  tooManyAttempts(<имя ограничителя>, <количество запросов>)
  ```

- `availableIn(<имя ограничителя>)` — возвращает время в секундах, спустя которое низкоуровневый ограничитель с заданным *именем* вновь позволит обрабатывать клиентские запросы:

  ```
  if (RateLimiter::tooManyAttempts('bb-list', 60))
      return response('Повторите попытку спустя ' .
                      RateLimiter::availableIn('bb-list') .
                      ' секунд', 429);
  ```

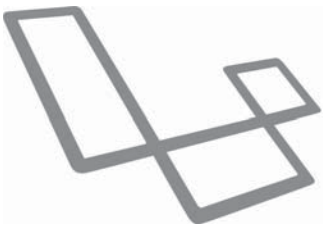
- `remaining()` — возвращает количество запросов, которые еще позволяет выполнить низкоуровневый ограничитель с заданным *именем*. Формат вызова такой же, как и у метода `tooManyAttempts()`;

- `hit(<имя ограничителя>[, <время ожидания>=60])` — увеличивает количество уже полученных запросов у низкоуровневого ограничителя с заданным *именем*. Как только количество запросов превысит ранее указанное у этого ограничителя, он будет ожидать истечения указанного *времени* в секундах, прежде чем разрешит обработку дальнейших запросов. Пример:

```
public function detail(Bb $bb) {
    if (RateLimiter::remaining('bb-detail-' . $bb->id, 60)) {
        RateLimiter::hit('bb-detail-' . $bb->id, 30);
        return view('bb-detail', ['bb' => $bb]);
    } else {
        return response('Слишком много запросов', 429);
    }
}
```

- `clear(<имя ограничителя>)` — обнуляет количество уже сделанных запросов, зарегистрированных низкоуровневым ограничителем с заданным *именем*:

```
class HomeController extends Controller {
    . . .
    public function update(Request $request, Bb $bb) {
        $bb->fill($request->all());
        $bb->save();
        RateLimiter::clear('bb-detail-' . $bb->id);
        return to_route('home');
    }
    . . .
{
```



ГЛАВА 20

Внедрение зависимостей, провайдеры и фасады

20.1. Внедрение зависимостей

Внедрением зависимостей называется автоматическое занесение требуемых значений в параметры методов, основываясь на совпадении имен и (или) типов этих параметров.

Наиболее характерный пример — получение объекта клиентского запроса в действии контроллера, для чего достаточно добавить в объявление действия параметр и указать у него в качестве типа класс запроса `Request`. Точно так же можно получить объект выбранной рубрики или объявления, указав в качестве типа параметра класс соответствующей модели (подробнее о внедрении моделей рассказывалось в *разд. 8.5.2*). Вот пример:

```
use Illuminate\Http\Request;
. . .
// В параметр request подсистема внедрения зависимостей занесет объект,
// представляющий текущий запрос, а в параметр rubric — объект выбранной
// рубрики
public function store(Request $request, Rubric $rubric) {
    $bb = new Bb($request->all());
    $bb->rubric_id = $rubric->id;
    . . .
}
```

20.1.1. Простейшие случаи внедрения зависимостей

Внедрение зависимостей работает с классами, как встроенными во фреймворк, так и написанными самим разработчиком.

В листинге 20.1 показан код класса `App\Repositories\BbRepository`, поддерживающего три метода:

- конструктор — в качестве первого параметра принимает объект клиентского запроса, заносимый подсистемой внедрения зависимостей, а в качестве второго, необязательного, — ключ рубрики (если не указан, извлекает его из URL-параметра `rubric`). Далее по полученному ключу ищет рубрику и присваивает ее защищенному свойству `rubric`;

- `rubric()` — возвращает рубрику из свойства `rubric`;
- `bbs()` — возвращает коллекцию объявлений, относящихся к рубрике из свойства `rubric`.

Листинг 20.1. Класс `App\Repositories\BbRepository`

```
namespace App\Repositories;
use Illuminate\Http\Request;
use App\Models\Rubric;
class BbRepository {
    private $rubric;

    public function __construct(Request $request, $rubric_id = null) {
        $rubric_id = $rubric_id ?? $request->route('rubric');
        $this->rubric = Rubric::find($rubric_id);
    }

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

    public function bbs() {
        return $this->rubric->bbs()->latest()->get();
    }
}
```

Чтобы получить объект этого класса, необходимо:

- в методе класса, вызываемом самим фреймворком (например, конструкторе или действии), если конструктору класса запрашиваемого объекта не нужно передавать параметры, — вставить, как было показано ранее, в объявление метода параметр с произвольным именем и указать у него в качестве типа класс, чей объект нужно получить:

```
use App\Repositories\BbRepository;
class MainController extends Controller {
    . . .
    public function rubric(BbRepository $bbr) {
        return view('rubric')->with(['rubric' => $bbr->rubric(),
                                     'bbs' => $bbr->bbs()]);
    }
}
```

- в остальных случаях — вызвав функцию-хелпер `resolve()`:

```
resolve(<путь к классу получаемого объекта>[,
        <массив со значениями параметров>=[]])
```

В качестве результата функция возвращает созданный объект. Пример:

```
// Получаем объект класса BbRepository, хранящий выбранную рубрику
$bbr = resolve(BbRepository::class);
$selectedRubric = $bbr->rubric();
```

Если конструктору класса, объект которого требуется получить, нужно передать какие-либо параметры, следует указать ассоциативный *массив со значениями параметров*. Ключи его элементов должны иметь те же имена, что и параметры конструктора класса, а значения элементов зададут значения соответствующих параметров. Пример:

```
// Получаем объект класса BbRepository, хранящий рубрику с ключом 9
$bbr = resolve(BbRepository::class, ['rubric_id' => 9]);
```

Вместо функции `resolve()` можно использовать функцию `app()`, имеющую тот же формат вызова;

- в методе любого провайдера — можно использовать способ более быстрый, нежели обращение к функции `resolve()`. Сначала следует получить объект подсистемы внедрения зависимостей, обратившись к свойству `app` провайдера, и вызвать у этого объекта метод `make()` или `makeWith()`. Оба метода имеют тот же формат вызова, что и функция `resolve()`, и полностью идентичны. Пример:

```
use App\Repositories\BbRepository;
class AppServiceProvider extends ServiceProvider {
    public function boot() {
        $rubric = $this->app->make(BbRepository::class,
                                   ['rubric_id' => 9]);
        . . .
    }
}
```

20.1.2. Управление внедрением зависимостей

Часто при создании объектов в процессе внедрения зависимостей нужно выполнять какие-либо дополнительные действия. Также бывают случаи, когда весь код сайта должен использовать один-единственный объект какого-либо класса (*синглтон*). Для таких ситуаций Laravel предоставляет развитые инструменты регистрации классов в подсистеме внедрения зависимостей.

Эти инструменты реализованы методами класса `Illuminate\Container\Container`. Единственный объект этого класса, представляющий саму подсистему внедрения зависимостей, можно получить из свойства `app`, поддерживаемого всеми провайдерами. Также эти методы можно вызвать у фасада `Illuminate\Support\Facades\App`.

Вызовы почти всех этих методов должны записываться в классе какого-либо провайдера — в теле общедоступного, не принимающего параметров метода `register()`. Если какой-то метод должен вызываться в другом методе провайдера, об этом будет сказано особо.

20.1.2.1. Простая регистрация классов и объектов

Далее приведены методы, просто регистрирующие классы и объекты в подсистеме внедрения зависимостей:

- `bind()` — регистрирует в подсистеме внедрения зависимостей класс с заданным *путем* и *анонимную функцию*, создающую объекты этого класса. Эта *функция* должна

принимать в качестве единственного параметра объект подсистемы внедрения зависимостей и возвращать готовый объект:

```
bind(<путь к классу>, <анонимная функция>[, <синглтон?>=false])
```

Пример регистрации класса `BbRepository` и функции, которая создает его объекты, хранящие рубрику с ключом 9:

```
use App\Repositories\BbRepository;
class AppServiceProvider extends ServiceProvider {
    public function register() {
        $this->app->bind(BbRepository::class, function($app) {
            // Первым параметром не забываем передать конструктору
            // класса объект текущего запроса
            return new BbRepository(request(), 9);
        });
    }
    . . .
}
```

Если параметру *синглтон* дать значение `true`, подсистема внедрения зависимостей создаст синглтон — единственный объект указанного класса, который и будет использоваться сайтом. Значение по умолчанию `false`, напротив, предписывает подсистеме создавать отдельный объект класса при каждом запросе на его получение. Пример:

```
$this->app->bind(BbRepository::class, function($app) {
    return new BbRepository(request(), 9);
}, true);
```

Вместо *пути к классу* можно указать произвольное строковое имя. Правда, получить сам объект можно будет лишь с помощью функций `resolve()` или `app()`, методов `make()` или `makeWith()`, в вызове которых следует указать имя, ранее заданное в методе `bind()`. Пример:

```
$this->app->bind('bbr', function($app) {
    return new BbRepository(request(), 9);
});
. . .
$bbr = resolve('bbr');
```

- `singleton(<путь к классу>[, <анонимная функция>=null])` — то же самое, что и `bind()`, только указывает подсистеме создать синглтон класса с указанным *путем*:

```
$this->app->singleton(BbRepository::class, function($app) {
    return new BbRepository(request(), 9);
});
```

Если *анонимная функция* не указана, подсистема внедрения зависимостей в дальнейшем просто создаст синглтон класса с указанным *путем* и будет выдавать его всем методам, которые его запросят:

```
$this->app->singleton(SingletonClass::class);
```

- `scoped()` — то же самое, что и `singleton()`, только создаваемый синглтон будет существовать только в течение обработки текущего клиентского запроса. При поступлении нового клиентского запроса старый синглтон будет уничтожен, а вместо него сгенерируется новый;
- `instance(<путь к классу>, <объект>)` — регистрирует в подсистеме уже созданный *объект* класса с указанным *путем*. В качестве результата возвращает заданный *объект*.

  Вызов этого метода может присутствовать как в методе `register()`, так и в методе `boot()` провайдера. Если *объектом* является запись модели или коллекция записей, его вызов следует помещать в методе `boot()`, поскольку, когда вызывается метод `register()` провайдера, модели еще не инициализированы. Пример:

```
$bbr = new BbRepository(request(), 9);
$this->app->instance(BbRepository::class, $bbr);
```

  Вместо пути к классу можно указать произвольное строковое имя, а вместо объекта — значение произвольного типа. Заданное значение далее можно получить, вызвав функцию `resolve()` или `app()`, метод `make()` или `makeWith()` с указанием имени, ранее заданного в методе `instance()`. Пример:

```
$this->app->instance('framework_name', 'Laravel');
. . .
$frameworkName = resolve('framework_name');
```

- `alias(<путь к классу>, <псевдоним>)` — задает *псевдоним* для класса с указанным *путем*. Получить объект этого класса можно, вызвав функцию `resolve()`или `app()`, метод `make()` или `makeWith()` с указанием *псевдонима*. Пример:

```
use Illuminate\Support\Facades\App;
App::alias(BbRepository::class, 'bbr');
. . .
$bbr = resolve('bbr');
```

- `bindIf()` — то же самое, что и `bind()`, только регистрирует класс лишь в том случае, если он ранее не был зарегистрирован. Формат вызова такой же, как и у метода `bind()`. Пример:

```
$this->app->bindIf(BbRepository::class, function($app) {
    return new BbRepository(request(), 9);
});
```

- `singletonIf()` — то же самое, что и `singleton()`, только регистрирует синглтон лишь в том случае, если он ранее не был зарегистрирован. Формат вызова такой же, как и у метода `singleton()`;
- `scopedIf()` — то же самое, что и `scoped()`, только регистрирует синглтон лишь в том случае, если он ранее не был зарегистрирован. Формат вызова такой же, как и у метода `singleton()`;
- `resolved(<путь к классу>|<объект>)` — возвращает `true`, если объект класса с указанным *путем* или заданный *объект* уже хотя бы раз были выданы подсистемой внедрения зависимостей, и `false` — в противном случае;

- `bound(<путь к классу>|<объект>)` — возвращает `true`, если класс с указанным *путем* или заданный *объект* уже зарегистрирован в подсистеме, или же класс с указанным *путем* имеет псевдоним, и `false` — в противном случае:

```
if (!$this->app->bound(BbRepository::class)) {
    $this->app->bind( ... );
}
```

- `has()` — то же самое, что и `bound()`;
- `isShared(<путь к классу>)` — возвращает `true`, если при регистрации класса с указанным *путем* подсистеме было предписано создавать синглтон, и `false` — в противном случае;
- `isAlias(<путь к классу>)` — возвращает `true`, если у класса с указанным *путем* был задан псевдоним, и `false` — в противном случае;
- `getAlias(<путь к классу>)` — возвращает псевдоним, заданный для класса с указанным *путем*. Если класс не имеет псевдонима, возвращается путь к нему.

20.1.2.2. Подмена классов и реализации

В вызовах методов `bind()`, `singleton()` и `scoped()` вместо *анонимной функции* можно указать путь к другому классу. В результате конструктор, запрашивающий у подсистемы внедрения зависимостей объект одного класса, получит объект другого класса.

> ***ПОДМЕНА РАБОТАЕТ ТОЛЬКО В КОНСТРУКТОРАХ КЛАССОВ...***
> ...но не в остальных их методах.

Это открывает следующие возможности:

- *подмена классов* — возврат объекта не запрашиваемого класса, а другого, который должен быть производным от него. В этом случае в вызове метода `bind()` первым параметром указывается путь к подменяемому классу, а вторым — путь к подменяющему. Пример:

```
$this->app->bind(SomeClass::class, AnotherClass::class);
```

  После чего любой конструктор, запросивший объект класса `SomeClass`, получит объект класса `AnotherClass`:

```
class ThirdClass {
    public function __construct(SomeClass $anotherClass) { ... }
    . . .
|
```

- *подмена реализации* — возврат объекта класса при указании в качестве типа интерфейса.

  Подмена реализации, в частности, позволит заменить какую-либо из подсистем Laravel другой — например, установленной в составе дополнительной библиотеки. Классы, представляющие подсистемы, должны реализовывать ключевые интерфейсы, называемые *контрактами*. Так, класс шаблонизатора реализует контракт `Illuminate\Contracts\View\Factory`, а класс подсистемы разграничения доступа — контракт `Illuminate\Contracts\Auth\Factory`.

В таком случае в вызове метода `bind()` первым параметром указывается путь к контракту, а вторым — путь к реализующему его классу:

```
$this->app->bind(\Illuminate\Contracts\View\Factory::class,
                 \SuperLibrary\SuperView::class);
```

После чего Laravel, запросив значение типа `Illuminate\Contracts\View\Factory`, получит объект класса `SuperLibrary\SuperView`, реализующий новый шаблонизатор.

Вместо контракта (и вообще интерфейса) можно указать абстрактный класс:

```
$this->app->bind(AbstractClass::class, ConcreteClass::class);
```

Альтернативой вызовам метода `bind()` может послужить использование общедоступного свойства `bindings` провайдера. Ему присваивается ассоциативный массив, ключами элементов которого служат пути к подменяемым классам, интерфейсам или абстрактным классам, а значениями — пути к подменяющим классам. Пример:

```
class AppServiceProvider extends ServiceProvider {
    public $bindings = [SomeClass::class => AnotherClass::class
                        SomeContract::class => WorkClass::class];
    . . .
}
```

Альтернативой использованию метода `singleton()` может стать применение аналогичного свойства `singletons`:

```
class AppServiceProvider extends ServiceProvider {
    . . .
    public $singletons = [
        SomeSingletonContract::class => WorkSingletonClass::class
    ];
    . . .
}
```

20.1.2.3. Гибкая подмена классов и реализации

При *гибкой подмене* тип выдаваемого объекта зависит от того, какой класс его запрашивает. Это позволяет выдавать конструкторам одного класса одни объекты, а конструкторам других классов — другие.

Для реализации гибкой подмены создается набор своего рода условий. В каждом условии указываются:

- класс, объект которого нужно выдать, сам выдаваемый объект или анонимная функция, создающая выдаваемый объект;
- классы, которые запрашивают выдаваемый объект;
- тип, который запрашивающие классы должны указать, чтобы получить этот объект.

Такое условие записывается по частям с использованием следующих методов:

- `when(`*`<путь к классу>`*`|`*`<массив путей к классам>`*`)` — вызывается непосредственно у объекта подсистемы внедрения зависимостей. Создает «пустое» условие и записывает в него *путь к классу*, который запрашивает объект, или *массив путей* к таким классам.

В качестве результата возвращает объект создаваемого условия. Все последующие методы вызываются у этого объекта:

- `needs(<запрашиваемый тип>)` — заносит в условие *запрашиваемый тип* обычно в виде пути к абстрактному классу или интерфейсу. Возвращает текущий объект условия;
- `give(<путь к классу>|<объект>|<анонимная функция>)` — задает выдаваемый *объект* непосредственно, в виде *пути* к его классу (тогда объект будет создан автоматически) или *анонимной функции*. Последняя не должна принимать параметров и должна возвращать созданный объект. Результата этот метод не возвращает.

Пример:

```
use Illuminate\Contracts\View\Factory as ViewContract;
. . .
$this->app->when(ComplexPageController::class)
          ->needs(ViewContract::class)
          ->give(function () {
              return new \SuperLibrary\SuperView;
            });
. . .
// Теперь контроллер ComplexPageController, запросив объект типа
// Illuminate\Contracts\View\Factory, получит объект класса
// SuperLibrary\SuperView
class ComplexPageController {
    protected $viewEngine;

    public function __construct(ViewContract $superView) {
        $this->viewEngine = $superView;
    }

    public function complexPage() {
        return $this->viewEngine->make('complex_page');
    }
}

// Контроллер VerySimplePageController, запросив объект того же типа,
// получит объект класса PoorLibrary\VerySimpleView
$this->app->when(VerySimplePageController::class)
          ->needs(ViewContract::class)
          ->give(\PoorLibrary\VerySimpleView::class);

// Остальные контроллеры получат объект стандартного шаблонизатора
```

20.1.2.4. Гибкая регистрация значений произвольного типа

Используя инструменты, описанные в *разд. 20.1.2.3*, в подсистеме внедрения зависимостей можно регистрировать значение произвольного типа, связав его с именем параметра метода, запрашивающего это значение. В этом случае в вызове метода `needs()` следует указать имя параметра обязательно с символом `$`. Пример:

```
$this->app->when(MainController::class)->needs('$siteName')
          ->give(function () {
              return config('app.name');
          });
. . .
public function __construct($siteName) { ... }
```

20.1.2.5. Переопределение регистрации

Еще можно дополнить выполненную ранее регистрацию каких-либо классов или объектов какой-либо новой функциональностью или полностью переопределить ее. Для этого служат следующие методы, поддерживаемые подсистемой внедрения зависимостей:

- `resolving([`*<путь к классу>*`, ]`*<анонимная функция>*`)` — задает *анонимную функцию*, которая будет выполнена сразу после создания объекта класса с указанным *путем*. *Анонимная функция* должна принимать в качестве параметров только что созданный объект и объект подсистемы внедрения зависимостей и не возвращать результат. Пример:

  ```
  $this->app->resolving(\SuperLibrary\SuperView::class,
                        function ($obj, $app) {
                            $obj->htmlVersion = '5.0';
                        });
  ```

  Если *путь к классу* не указан, *анонимная функция* будет выполняться после создания объекта любого класса (что может пригодиться, например, для журналирования или отладки):

  ```
  $this->app->resolving(function ($obj, $app) { dump($obj); });
  ```

- `afterResolving()` — то же самое, что и `resolving()`, только заданная в его вызове *анонимная функция* выполняется после исполнения *функций* из вызовов метода `resolving()`. Практически всегда используется при отладке и журналировании;

- `extend()` — то же самое, что и `resolving()`, только заданная *анонимная функция* должна возвращать другой объект того же класса, к которому принадлежит созданный объект, или класса, реализующего тот же интерфейс:

  ```
  $this->app->extend(\SuperLibrary\SuperView::class,
                     function ($obj, $app) {
                         return new \MegaSuperLibrary\MegaSuperView;
                     });
  ```

20.1.2.6. Вызов методов и функций, в которых используется внедрение зависимостей

Если конструктор класса использует внедрение зависимостей, чтобы получить через какой-либо параметр нужный ему объект (например, конструктор класса `BbRepository` из листинга 20.1), создать объект такого класса оператором `new` невозможно. При использовании этого оператора внедрение зависимостей не выполнится, конструктор не получит нужный объект и завершится с ошибкой.

Для создания объектов таких классов следует применять способы, описанные в *разд. 20.1.1*, например:

```
$bbr = resolve(BbRepository::class);
```

Точно так же не получится напрямую вызвать у объекта метод, использующий внедрение зависимостей. Для этого нужно применить прием, описываемый далее.

Сначала следует получить объект подсистемы внедрения зависимостей. Его можно извлечь из свойства `app` провайдера, получить вызовом функции `app()` без параметров или обратиться к фасаду `App`.

Метод `call()` этого объекта выполняет вызов метода с заданным *именем* у указанного *объекта*, возможно, с передачей значений параметров, задаваемых явно, из указанного *ассоциативного массива*:

```
call(<массив с объектом и именем метода>[,
     <ассоциативный массив со значениями параметров>=[]])
```

Пример:

```
// Вызываемый метод
class SomeClass {
    public function someMethod(Request $request, $id, $mode = 'i') {
        . . .
    }
}
. . .
$object = new SomeClass;
// Вызов метода
app()->call([$object, 'someMethod'], ['id' => 2, 'mode' => 'r']);
```

Можно вызвать статический метод, указав в *массиве* вместо *объекта* путь к классу:

```
// Вызываемый метод
class SomeClass2 {
    public static function someStaticMethod(Request $request, $id) {
        . . .
    }
}
. . .
// Вызов статического метода
app()->call([SomeClass2::class, 'someStaticMethod'], ['id' => 2]);
```

Первым параметром также можно указать строку формата *<путь к классу>*::*<имя статического метода>*:

```
app()->call('App\\Classes\\SomeClass2::someStaticMethod', ['id' => 2]);
```

Метод `call()` можно использовать, чтобы создать объект класса с заданным *путем* и сразу же вызвать у него метод с указанным *именем*. Для этого первым параметром указывается строка формата *<путь к классу>*@*<имя метода>*, а вторым — *массив со значениями параметров*:

```
app()->call('App\\Classes\\SomeClass@someMethod', ['id' => 2]);
```

При этом заданные в *массиве* параметры будут переданы и конструктору, и указанному в вызове методу.

Для таких случаев применим альтернативный формат вызова метода:

```
call(<путь к классу>, <ассоциативный массив со значениями параметров>,
                      <имя вызываемого метода>)
```

Пример:

```
app()->call('SomeClass', ['id' => 2], 'someMethod');
```

Чтобы вызвать функцию, использующую внедрение зависимостей, в первом параметре метода `call()` указывается имя функции в виде строки, а во втором — *массив со значениями параметров*:

```
// Вызываемая функция
function someFunc(Request $request, $id) { ... }
. . .
// Вызов функции
$result = app()->call('someFunc', ['id' => 4]);
```

В первом параметре метода `call()` также можно указать анонимную функцию, получающую параметры посредством внедрения зависимостей, — и эта функция будет успешно вызвана:

```
app()->call(function (Request $request, $id) {
    . . .
}, ['id' => 4]);
```

20.1.2.7. Подмена методов

Наконец, можно подменить какой-либо метод любого класса анонимной функцией, которая будет вызываться вместо него. Это выполняет метод `bindMethod()` подсистемы внедрения зависимостей:

```
bindMethod(<массив с путем к классу и именем метода>|↳
           <обозначение класса и метода>, <анонимная функция>)
```

Обозначение класса и метода записывается в виде строки формата *<путь к классу>@<имя подменяемого метода>*. *Анонимная функция* должна принимать два параметра: объект, которому принадлежит вызываемый подмененный метод, и объект подсистемы внедрения зависимостей. Пример:

```
$this->app->bindMethod([SomeClass::class, 'someMethod'],
                       function ($obj, $app) {
                           . . .
                       });

// Здесь вместо метода someClass() реально будет выполнена анонимная
// функция, указанная в вызове метода bindMethod()
app()->call('SomeClass', ['id' => 2], 'someMethod');
```

Метод `hasMethodBindings(<имя метода>)` возвращает `true`, если метод с указанным *именем* был подменен, и `false` — в противном случае.

Действия контроллера подменить невозможно

Подмена работает только с обычными методами, вызываемыми явно.

20.2. Провайдеры

Провайдер — программный модуль, задающий режим работы и некоторые параметры какой-либо из подсистем фреймворка. Провайдеры выполняются одномоментно при инициализации сайта.

20.2.1. Список провайдеров, используемых веб-сайтом

Список провайдеров, которые используются сайтом, указывается в вида массива, присвоенного рабочей настройке `providers` из модуля config\app.php.

Массив провайдеров можно условно разделить на три части: провайдеры, находящиеся в составе самого фреймворка (изначально там содержатся все провайдеры Laravel), присутствующие в дополнительных библиотеках (изначально эта часть пуста) и создаваемые в составе проекта (их четыре, плюс один закомментированный). Части массива помечены соответствующими комментариями.

Многие провайдеры обслуживают подсистемы, не являющиеся обязательными для функционирования сайта (например, подсистемы кеширования, рассылки почты или взаимодействия с Redis). Если такая подсистема не используется сайтом, соответствующий провайдер может быть удален из массива или закомментирован — это сэкономит оперативную память.

Далее приведен алфавитный список всех провайдеров, имеющихся в составе фреймворка, с указанием инициализируемых ими подсистем:

- `Illuminate\Auth\AuthServiceProvider` — подсистема разграничения доступа;
- `Illuminate\Broadcasting\BroadcastServiceProvider` — подсистема *вещания* (см. *главу 31*). Если таковая не используется, провайдер можно убрать из массива;
- `Illuminate\Bus\BusServiceProvider` — подсистема отложенных заданий (см. *главу 25*). Если отложенные задания не используются, провайдер можно удалить из массива;
- `Illuminate\Cache\CacheServiceProvider` — подсистема кеширования (см. *главу 29*). Если кеширование на стороне сервера не задействовано, провайдер можно удалить из массива;
- `Illuminate\Foundation\Providers\ConsoleSupportServiceProvider` — подсистема консоли Laravel;
- `Illuminate\Cookie\CookieServiceProvider` — подсистема обработки cookie (см. *главу 26*), используемая средствами разграничения доступа;
- `Illuminate\Database\DatabaseServiceProvider` — подсистема работы с базами данных;
- `Illuminate\Encryption\EncryptionServiceProvider` — подсистема шифрования, используемая средствами разграничения доступа;

- `Illuminate\Filesystem\FilesystemServiceProvider` — подсистема обработки выгруженных файлов. Если сайт не сохраняет выгруженные посетителями файлы, провайдер можно удалить;
- `Illuminate\Foundation\Providers\FoundationServiceProvider` — ряд ключевых подсистем, используемых, в частности, подсистемами обработки ошибок и валидации;
- `Illuminate\Hashing\HashServiceProvider` — подсистема хеширования, используемая несколькими подсистемами, например разграничения доступа и обработки выгруженных файлов;
- `Illuminate\Mail\MailServiceProvider` — подсистема отправки электронной почты (см. *главу 23*). Если сайт не рассылает почту, провайдер можно удалить;
- `Illuminate\Notifications\NotificationServiceProvider` — подсистема отправки оповещений (см. *главу 24*). Если сайт не рассылает оповещения, провайдер можно удалить;
- `Illuminate\Pagination\PaginationServiceProvider` — подсистема пагинации. Если пагинация не используется, провайдер можно убрать из массива;
- `Illuminate\Pipeline\PipelineServiceProvider` — ключевая подсистема, используемая, в частности, подсистемой посредников;
- `Illuminate\Queue\QueueServiceProvider` — подсистема очередей;
- `Illuminate\Redis\RedisServiceProvider` — подсистема взаимодействия с СУБД Redis. Если эта СУБД не используется, провайдер можно удалить;
- `Illuminate\Auth\Passwords\PasswordResetServiceProvider` — подсистема сброса пароля. Если на сайте не реализован сброс пароля, провайдер можно удалить;
- `Illuminate\Session\SessionServiceProvider` — подсистема серверных сессий (см. *главу 26*), используемая несколькими другими подсистемами;
- `Illuminate\Translation\TranslationServiceProvider` — подсистема локализации (см. *главу 28*);
- `Illuminate\Validation\ValidationServiceProvider` — подсистема валидации;
- `Illuminate\View\ViewServiceProvider` — шаблонизатор.

Алфавитный список всех провайдеров, изначально создаваемых в составе проекта:

- `App\Providers\AppServiceProvider` — задает специфичные для конкретного сайта параметры подсистем, за исключением разграничения доступа, вещания, событий (см. *главу 22*) и маршрутизатора;
- `App\Providers\AuthServiceProvider` — используется для создания гейтов (см. *разд. 13.8.2*) и связывания политик с моделями (см. *разд. 13.8.3*);
- `App\Providers\BroadcastServiceProvider` — регистрирует в маршрутизаторе маршруты, ведущие на каналы вещания (см. *главу 31*). Изначально закомментирован;
- `App\Providers\EventServiceProvider` — привязывает обработчики к событиям (см. *главу 22*);
- `App\Providers\RouteServiceProvider` — регистрирует в маршрутизаторе веб- и API-маршруты и создает ограничители скорости запросов.

20.2.2. Создание своих провайдеров

В процессе программирования сложных сайтов код провайдеров, изначально создаваемых в составе проекта (в первую очередь это относится к «универсальному» провайдеру `AppServiceProvider`), может чрезмерно вырасти в объеме. В таком случае удобнее создать свои провайдеры и вынести часть кода в них.

Новый провайдер создается командой:

```
php artisan make:provider <имя класса провайдера>
```

Созданный провайдер объявляется в пространстве имен `App\Providers` и становится производным от класса `Illuminate\Support\ServiceProvider`. Он содержит два общедоступных, не принимающих параметров метода, изначально «пустых»:

- `register()` — выполняется в начале инициализации сайта, когда запускаются его подсистемы. Служит для записи кода, управляющего внедрением зависимостей (см. *разд. 20.1.2*);
- `boot()` — выполняется после инициализации всех подсистем сайта. Служит для записи всего остального кода, обычно указываемого в провайдерах (регистрация разделяемых значений, составителей значений и пр.).

В листинге 20.2 показан код вновь созданного провайдера `App\Providers\MyServiceProvider`, регистрирующего три значения в подсистеме внедрения зависимостей и разделяемое значение — в шаблонизаторе.

Листинг 20.2. Код провайдера `App\Providers\MyServiceProvider`

```
namespace App\Providers;
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\View;
use App\Http\Controllers\MainController;
use App\Repositories\BbRepository;
use App\Repositories\UserRepository;
class MyServiceProvider extends ServiceProvider {
    public function register() {
        $this->app->bind(UserRepository::class, function($app) {
            return new UserRepository(['active' => true]);
        });
        $this->app->bind('bbr', BbRepository::class);
        $this->app->when(MainController::class)->needs('$siteName')
            ->give(function () {
                return config('app.name');
            });
    }

    public function boot() {
        View::share('copyright', '© команда разработчиков');
    }
}
```

В методе `boot()` провайдеров работает внедрение зависимостей:

```
use App\Repositories\BbRepository;
. . .
public function boot(BbRepository $bbr) {
    . . .
}
```

Созданный провайдер следует добавить в массив, хранящийся в рабочей настройке `app.providers`, иначе Laravel не узнает о его существовании:

```
'providers' => [
    . . .
    App\Providers\MyServiceProvider::class,
],
```

Если провайдер содержит лишь код, управляющий внедрением зависимостей, его можно пометить как *обрабатываемый по запросу*. Такой провайдер выполняется лишь тогда, когда фреймворку требуется выдать объект зарегистрированного в нем класса.

Чтобы пометить провайдер как обрабатываемый по запросу, следует:

1. Реализовать в классе провайдера интерфейс `Illuminate\Contracts\Support\DeferrableProvider`.
2. Объявить в классе провайдера общедоступный, не принимающий параметров метод `provides()`, который должен возвращать массив имен, под которыми провайдер регистрирует значения в подсистеме внедрения зависимостей.

Так, чтобы превратить провайдер `MyServiceProvider` (см. листинг 20.2) в обрабатываемый по запросу, в его код следует внести следующие правки:

```
use Illuminate\Contracts\Support\DeferrableProvider;
class MyServiceProvider extends ServiceProvider
      implements DeferrableProvider {
    . . .
    public function provides() {
        return [BbRepository::class, 'bbr', '$siteName'];
    }
}
```

20.3. Фасады

Фасад — это класс, предоставляющий удобный доступ к объекту (как правило, синглтону), который реализует одну из ключевых подсистем фреймворка и дополнительных библиотек: маршрутизатор, построитель запросов, шаблонизатор, создатель миниатюр и т. п. Фасад транслирует сделанный у него вызов статического метода в вызов обычного метода представляемого им объекта подсистемы.

Рабочая настройка `aliases` из модуля config\app.php хранит список всех фасадов, доступных для использования в шаблонах. Этот список представляет собой ассоциативный массив, ключами элементов которого являются обозначения фасадов, указываемые в коде шаблонов, а значениями элементов — пути к классам соответствующих фасадов. Изначально обозначения фасадов практически всегда совпадают с именами их классов.

Массив фасадов, присваиваемый рабочей настройке `app.aliases`, формируется путем объединения:

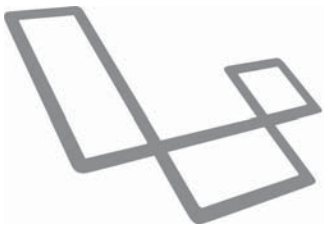
- коллекции фасадов, входящих в состав фреймворка, — получаемой вызовом статического метода `defaultAliases()` класса `Illuminate\Support\Facades\Facade` (который является базовым классом для всех фасадов);
- массива фасадов, добавленных разработчиком сайта, — его присоединение к полученной ранее коллекции выполняется вызовом метода `merge()` у последней. Массив добавленных фасадов, передаваемый методу `merge()`, изначально «пуст».

Результат объединения коллекций переводится в обычный ассоциативный массив вызовом метода `toArray()` и присваивается упомянутой ранее рабочей настройке.

Фасады, добавляемые разработчиком сайта, могут как поставляться в составе дополнительных библиотек, так и быть написаны своим разработчиком. Но рассказ о написании своих фасадов выходит за рамки этой книги.

Пример добавления в список двух дополнительных фасадов:

```
'aliases' => Facade::defaultAliases()->merge([
    'Facade1' => SomeLibrary\Facades\Facade1::class,
    'Facade2' => OtherLibrary\Support\Facades\Facade2::class,
])->toArray(),
```



ГЛАВА 21

Посредники

Посредник — это программный модуль, выполняющий предварительную обработку запроса после его получения от клиента и перед передачей его контроллеру и (или) окончательную обработку ответа после его формирования контроллером и перед отправкой клиенту.

21.1. Посредники, используемые веб-сайтом

Списки посредников, зарегистрированных в проекте, записываются в защищенных свойствах «корневого» класса маршрутизатора `App\Http\Kernel`: `middleware` (связываемые со всеми маршрутами), `middlewareGroups` (список групп посредников с входящими в их состав посредниками) и `routeMiddleware` (связываемые с маршрутами явно). Более подробно эти списки рассматривались в *разд. 8.1*.

Вот список посредников, связываемых со всеми маршрутами (свойство `middleware`):

- `App\Http\Middleware\TrustHosts` — задает список доверенных хостов (подробнее о его настройке будет рассказано в *главе 35*). Изначально закомментирован;
- `App\Http\Middleware\TrustProxies` — задает список доверенных прокси-серверов (подробности — в *главе 35*). Если прокси-серверы не используются, посредник может быть удален из списка — это немного повысит производительность сайта;
- `Illuminate\Http\Middleware\HandleCors` — реализует технологию CORS (Cross-Origin Resource Sharing, совместное использование ресурсов между разными источниками), обеспечивающую получение данных со сторонних сайтов (см. *главу 30*). Если разрабатываемый веб-сайт не включает в себя бэкенд, обслуживающий фронтенды с других сайтов, посредник может быть удален;
- `App\Http\Middleware\PreventRequestsDuringMaintenance` — блокирует доступ к сайту, если тот находится в режиме обслуживания (будет описан в *главе 35*), и задает список путей к страницам, которые должны быть доступны посетителям даже в этом случае. Если на сайте не предусматривается режим обслуживания, посредник можно удалить;
- `Illuminate\Foundation\Http\Middleware\ValidatePostSize` — проверяет, не превышает ли объем данных, отправленных из веб-формы методом POST, заданную в настрой-

ках PHP величину, и, если превышает, выдает ошибку 413 (слишком большой объем данных POST). Если такую проверку проводить не нужно (например, если сайт принципиально не принимает данные по методу POST), может быть удален;

- `App\Http\Middleware\TrimStrings` — удаляет из значений, отправленных из веб-формы, начальные и конечные пробелы. Если такую операцию выполнять не требуется, посредник можно удалить;
- `Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull` — преобразует «пустые» строки, отправленные из веб-формы, в значения `null`. Если такую операцию выполнять не требуется, может быть удален.

Список групп посредников, созданных в массиве из свойства `middlewareGroups`, вместе с входящими в их состав посредниками:

- `web`:
  - `App\Http\Middleware\EncryptCookies` — шифрует cookie (см. *главу 26*);
  - `Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse` — обеспечивает постановку cookie в очередь для отправки в составе ответа перед его созданием (о cookie речь пойдет в *главе 26*). Если функциональность очереди cookie не используется, посредник можно удалить;
  - `Illuminate\Session\Middleware\StartSession` — запускает серверную сессию при получении очередного запроса;
  - `Illuminate\View\Middleware\ShareErrorsFromSession` — создает в контексте каждого шаблона переменную `errors`, хранящую список ошибок ввода. Если возможность ошибочного ввода принципиально исключена (например, если для ввода данных применяются лишь флажки, переключатели и списки), посредник может быть удален;
  - `App\Http\Middleware\VerifyCsrfToken` — сверяет электронные жетоны безопасности, применяемые для защиты от атак CSRF (см. *разд. 19.3*);
  - `Illuminate\Routing\Middleware\SubstituteBindings` — реализует внедрение моделей (см. *разд. 8.5.2*). Если внедрение моделей не используется, посредник можно удалить.
- `api`:
  - `Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful` — при использовании для разграничения доступа библиотеки Laravel Sanctum (в этой книге не описывается) обеспечивает хранение данных о пользователе на стороне сервера — в сессиях. Изначально закомментирован;
  - `throttle:api` — указывает управление скоростью запросов посредством ограничителя `api` (см. *разд. 19.4.2*);
  - `SubstituteBindings` — описан ранее.

Список посредников (обозначения указаны в скобках), содержащихся в массиве из свойства `routeMiddleware`:

- `App\Http\Middleware\Authenticate` (`auth`) — перенаправляет гостей на страницу входа. Связывается с маршрутами на страницы, которые должны быть доступны только для зарегистрированных пользователей, выполнивших вход;

- `Illuminate\Auth\Middleware\AuthenticateWithBasicAuth` (`auth.basic`) — предписывает веб-обозревателю перед переходом на страницу выполнить базовую аутентификацию.

  Базовая аутентификация реализуется теми же механизмами Laravel, что и обычный вход. Однако при базовой аутентификации адрес электронной почты и пароль пользователя для выполнения входа вводятся не на специально предназначенной для того веб-странице, а в диалоговом окне, выводимом самим веб-обозревателем. Выход с сайта выполняется после закрытия веб-обозревателя.

  Пример использования посредника:

  ```
  Route::resource('rubrics', 'RubricController')->middleware('auth.basic');
  ```

  > ***Совет***
  >
  > На взгляд автора, базовую аутентификацию имеет смысл использовать лишь на первых этапах разработки сайта, при настройке подсистемы разграничения доступа. После чего базовую аутентификацию следует заменить на обычную, со страницами входа и выхода;

- `Illuminate\Session\Middleware\AuthenticateSession` (`auth.session`) — подписывает серверную сессию текущим паролем пользователя. Используется при реализации завершения пользовательских сессий на других устройствах после изменения пароля (см. *разд. 19.1.7*);

- `Illuminate\Http\Middleware\SetCacheHeaders` (`cache.headers`) — управляет кешированием на стороне клиента (см. *главу 29*);

- `Illuminate\Auth\Middleware\Authorize` (`can`) — используется для разграничения доступа в маршрутах на основе политик (см. *разд. 13.8.3.2*);

- `App\Http\Middleware\RedirectIfAuthenticated` (`guest`) — перенаправляет зарегистрированных пользователей, выполнивших вход, по интернет-адресу из константы `HOME` провайдера `RouteServiceProvider` (см. *разд. 8.1*). Связывается с маршрутами на страницы, которые должны быть доступны только гостям;

- `Illuminate\Auth\Middleware\RequirePassword` (`password.confirm`) — перенаправляет пользователя на страницу подтверждения пароля (см. *разд. 13.10*);

- `Illuminate\Routing\Middleware\ValidateSignature` (`signed`) — выполняет проверку подписанных интернет-адресов (речь о них пойдет в *главе 26*);

- `Illuminate\Routing\Middleware\ThrottleRequests` (`throttle`) — управляет скоростью запросов (см. *разд. 19.4*);

- `Illuminate\Auth\Middleware\EnsureEmailIsVerified` (`verified`) — перенаправляет пользователя на страницу подтверждения существования адреса электронной почты, если адрес не был подтвержден ранее (см. *разд. 13.12*).

В любой из этих списков можно добавить посредники, написанные самим разработчиком сайта или присутствующие в составе дополнительных библиотек.

21.1.1. Управление очередностью выполнения посредников

При поступлении очередного клиентского запроса и его «прохождении» по маршруту посредники исполняются в следующем порядке:

1. Связываемые со всеми маршрутами (заданные в свойстве `middleware` класса `App\Http\Kernel`).
2. Связываемые с группой, к которой принадлежит маршрут (свойство `middlewareGroups` того же класса).
3. Связанные с маршрутом в списке маршрутов, — в том порядке, в котором они приводятся в вызовах метода `middleware()` (см. *разд. 8.6*).
4. Связываемые с контроллером, на который указывает маршрут, — также в порядке их указания в вызовах метода `middleware()` (см. *разд. 9.1.2.4*).

Можно указать другой порядок выполнения посредников. Для этого достаточно объявить в классе `App\Http\Kernel` защищенное свойство `middlewarePriority` и присвоить ему массив путей к классам посредников, выстроенным в нужном порядке. Пример:

```
class Kernel extends HttpKernel {
    . . .
    protected $middlewarePriority = [
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \Illuminate\Routing\Middleware\ThrottleRequests::class,
        \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        \Illuminate\Auth\Middleware\Authorize::class,
    ];
}
```

21.1.2. Параметры посредников

Посредники могут принимать параметры. Их значения указываются в формате:

<обозначение посредника>:*<параметр 1>*,*<параметр 2>*, ... *<параметр n>*

Пример указания двух параметров у посредника `ThrottleRequests` (обозначение `throttle`):

```
Route::get('rubrics', ...)->middleware('throttle:30,2');
```

21.2. Написание своих посредников

Если требуется производить свою собственную обработку получаемых запросов и (или) отправляемых ответов, можно написать оригинальный посредник.

21.2.1. Как исполняется посредник?

Каждый посредник, зарегистрированный в списках из класса `App\Http\Kernel`, в течение обработки каждого клиентского запроса исполняется дважды:

1. Сразу после получения клиентского запроса и перед передачей его контроллеру. При этом посредники исполняются в порядке, описанном в *разд. 21.1.1*.
2. Сразу после генерирования контроллером ответа и перед отправкой его клиенту. В этом случае посредники исполняются в порядке, противоположном описанному в *разд. 21.1.1*.

   На этой, второй, фазе работы посредник имеет доступ как к запросу, так и к ответу, что может пригодиться в ряде случаев.

Эти сведения потребуются для понимания принципов, согласно которым пишутся посредники.

21.2.2. Создание посредников

Новый «пустой» класс посредника создается подачей команды:

```
php artisan make:middleware <имя класса посредника>
```

Класс посредника объявляется в пространстве имен `App\Http\Middleware` и не является производным от какого бы то ни было класса (хотя его можно сделать подклассом другого, уже существующего во фреймворке посредника).

Класс посредника должен содержать общедоступный метод `handle()`, принимающий следующие параметры:

- объект клиентского запроса;
- анонимную функцию, представляющую посредник, который будет выполнен сразу после текущего;
- параметры, хранящие значения параметров посредника (см. *разд. 21.1.2*).

Вся логика посредника реализуется в его методе `handle()`. Ее можно разделить на четыре части:

1. Обработка полученного с первым параметром клиентского запроса (может отсутствовать).
2. Вызов функции, полученной со вторым параметром. Единственным параметром ей следует передать объект клиентского запроса. Возвращенный ею результат — объект сгенерированного контроллером ответа — необходимо сохранить в какой-либо переменной.
3. Обработка полученного на *шаге 2* серверного ответа (может отсутствовать).
4. Возврат объекта серверного ответа из метода `handle()` в качестве результата.

В листинге 21.1 показан код своего рода шаблона, по которому программируются посредники.

Листинг 21.1. Шаблон, согласно которому пишутся посредники

```
namespace App\Http\Middleware;
use Closure;
use Illuminate\Http\Request;
```

```
class MyMiddleware {
    public function handle(Request $request, Closure $next) {
        // Обработка запроса request

        $response = $next($request);

        // Обработка ответа response

        return $response;
    }
}
```

В листинге 21.2 показан код посредника `App\Http\Middleware\MyMiddleware`, который, если запрос пришел от веб-обозревателя Google Chrome, вставляет в отправляемый ответ заголовок `X-Chrome-Greeting` со значением `'Hello, Chrome!'`.

Листинг 21.2. Код посредника `App\Http\Middleware\MyMiddleware`

```
namespace App\Http\Middleware;
use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Str;
class MyMiddleware {
    public function handle(Request $request, Closure $next) {
        $isChrome = Str::contains($request->header('User-Agent'),
                                  'Chrome');
        $response = $next($request);
        if ($isChrome)
            $response->header('X-Chrome-Greeting', 'Hello, Chrome!');
        return $response;
    }
}
```

Созданный посредник необходимо зарегистрировать в одном из списков класса `App\Http\Kernel`:

```
class Kernel extends HttpKernel {
    protected $middleware = [
        . . .
        \App\Http\Middleware\MyMiddleware::class,
    ];
    . . .
}
```

В классе посредника можно объявить любые другие методы, в том числе конструктор. В конструкторе посредника действует внедрение зависимостей.

Посредник может выполнять перенаправление на другие страницы, при этом код, производящий перенаправление, должен исполняться на *шаге 1*.

В листинге 21.3 показан код посредника `App\Http\Middleware\UserMiddleware`, пускающего на страницу только пользователя, чье имя указано в параметре этого посредника,

а всех остальных пользователей перенаправляющий по интернет-адресу из константы `HOME` провайдера `RouteServiceProvider`.

Листинг 21.3. Код посредника `App\Http\Middleware\UserMiddleware`

```
namespace App\Http\Middleware;
use Closure;
use Illuminate\Http\Request;
use App\Providers\RouteServiceProvider;
class UserMiddleware {
    public function handle(Request $request, Closure $next, $userName) {
        if ($request->user()->name != $userName)
            return redirect(RouteServiceProvider::HOME);
        return $next($request);
    }
}
```

Пример использования этого посредника:

```
class Kernel extends HttpKernel {
    . . .
    protected $routeMiddleware = [
        . . .
        'username' => \App\Http\Middleware\UserMiddleware::class,
    ];
}
. . .
// Теперь попасть в раздел рубрик сможет только пользователь admin
Route::resource('rubrics', RubricController::class)
     ->middleware(['auth', 'username:admin']);
```

Очень часто в качестве серверного ответа клиенту отсылается страница, сгенерированная на основе шаблона. Перед генерированием страницы посредник может добавить в контекст шаблона произвольный набор переменных, воспользовавшись методом `share()` фасада `View` (см. *разд. 11.8.1*). Пример:

```
use Illuminate\Support\Facades\View;
. . .
public function handle($request, Closure $next) {
    . . .
    View::share('copyright', '© команда разработчиков');
    $response = $next($request);
    . . .
}
```

21.2.3. Посредники с завершающими действиями

Иногда бывает необходимо выполнить какие-либо завершающие действия (например, занести запись в журнал) уже после того, как сгенерированный контроллером ответ отправится клиенту. Эти действия можно выполнить в общедоступном методе

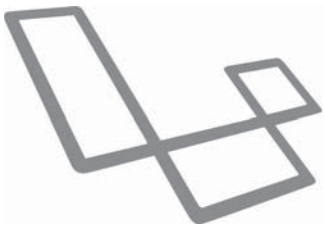
`terminate()` класса посредника, который должен принимать в качестве параметров объекты запроса и ответа. Пример:

```
class MyMiddleware {
    public function handle(Request $request, Closure $next) { ... }

    public function terminate($request, $response) {
        new LogEntry($request);
        new LogEntry($response);
    }
}
```

По умолчанию фреймворк создает два объекта посредника: у первого он вызывает метод `handle()`, а у второго — метод `terminate()`. Если нужно вызывать оба метода у одного и того же объекта (что может пригодиться в случае, если метод `handle()` сохраняет в свойствах объекта какие-либо данные, далее используемые методом `terminate()`), следует зарегистрировать класс посредника в подсистеме внедрения зависимостей и указать, чтобы она создавала лишь один объект этого класса (синглтон, подробности — в *разд. 20.1.2.1*). Пример:

```
class AppServiceProvider extends ServiceProvider {
    public function register() {
        . . .
        $this->app->singleton(App\Http\Middleware\UserMiddleware::class);
    }
    . . .
}
```



ГЛАВА 22

События и их обработка

Событие — это сообщение, генерируемое фреймворком при наступлении какого-либо условия: добавления записи в базу данных, удаления записи, выполнения пользователем входа на сайт и др. *Обработчик события* — это программный код, выполняемый при возникновении события определенного типа. К одному событию можно привязать произвольное количество обработчиков.

Обработка событий позволяет заданным образом отреагировать на что-либо, произошедшее во фреймворке. Например, можно заносить в журнал запись, когда очередной пользователь входит на сайт, или удалять выгруженный файл при удалении записи из базы данных.

Также можно объявить свои события и генерировать их в нужные моменты времени.

22.1. События-классы

Событие-класс представляется классом. Такие события генерируются всеми подсистемами фреймворка, за исключением подсистемы моделей.

22.1.1. Обработка событий-классов: слушатели

Слушатель (listener) — наиболее простой из обработчиков, позволяющий обрабатывать события только одного типа. Слушатель может быть реализован в виде класса (*слушатель-класс*) или анонимной функции (*слушатель-функция*).

22.1.1.1. Создание слушателей-классов

Новый слушатель-класс генерируется командой:

```
php artisan make:listener <имя класса слушателя> ↳
[--event=<путь к классу обрабатываемого события>] [--queued]
```

По умолчанию создается класс обычного слушателя, не настроенный для обработки какого-либо конкретного события.

Поддерживаются следующие командные ключи:

- `--event` — сразу настраивает создаваемый слушатель для обработки события, представляемого классом с указанным *путем*.

Пример создания слушателя `UserAttemptListener`, обрабатывающего событие `Illuminate\Auth\Events\Attempting`:

```
php artisan make:listener UserAttemptListener ↳
--event=Illuminate\Auth\Events\Attempting
```

- `--queued` — создает отложенный слушатель (такие слушатели будут описаны в *главе 25*).

Класс слушателя объявляется в пространстве имен `App\Listeners` (соответствующая папка создается автоматически) и не является ничьим подклассом. В классе слушателя изначально присутствуют два общедоступных метода:

- конструктор — изначально не принимающий параметров и «пустой».

  В конструкторах слушателей работает внедрение зависимостей;

- `handle()` — принимающий в качестве единственного параметра объект события.

  Если при создании слушателя командой `make:listener` утилиты artisan был указан ключ `--event`, в качестве типа этого параметра будет подставлен класс события, путь к которому был указан в этом ключе. Если же ключ `--event` не был указан, параметр не будет иметь никакого типа.

В листинге 22.1 показан код слушателя `App\Listeners\UserAttemptListener`, обрабатывающего событие `Illuminate\Auth\Events\Attempting`, которое генерируется при попытке пользователя выполнить вход на сайт. Этот обработчик записывает адреса электронной почты и пароли, введенные в веб-форму входа, в файл storage\app\attempts.log.

Листинг 22.1. Код слушателя-класса `App\Listeners\UserAttemptListener`

```
namespace App\Listeners;
use Illuminate\Auth\Events\Attempting;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Facades\Storage;
class UserAttemptListener {
    public function __construct() { }

    public function handle(Attempting $event) {
        $s = 'email: ' . $event->credentials['email'] .
             ', password: ' . $event->credentials['password'];
        Storage::disk('local')->append('attempts.log', $s);
    }
}
```

Данные, введенные в веб-форму входа, можно извлечь из ассоциативного массива, хранящегося в свойстве `credentials` объекта события (более подробно классы событий будут рассмотрены далее).

Ранее говорилось, что к одному событию можно привязать произвольное количество обработчиков: слушателей или подписчиков, описываемых далее. Эти обработчики будут выполняться один за другим в том порядке, в котором была выполнена их привязка.

Если после выполнения какого-либо слушателя следует отменить исполнение последующих обработчиков, из метода `handle()` следует вернуть значение `false`:

```
public function handle(Attempting $event) {
    . . .
    // Отменяем выполнение последующих обработчиков
    return false;
}
```

22.1.1.2. Явная привязка слушателей-классов к событиям

Привязка слушателей-классов к событиям выполняется в ассоциативном массиве, присвоенном защищенному свойству `listen` провайдера `App\Providers\EventServiceProvider`. Ключами элементов этого массива должны быть пути к классам событий, а элементами — массивы путей к классам привязываемых слушателей.

Изначально в этом массиве содержится слушатель `Illuminate\Auth\Listeners\SendEmailVerificationNotification`, обрабатывающий событие `Illuminate\Auth\Events\Registered`, которое возникает после регистрации нового пользователя. Этот слушатель отправляет зарегистрированному пользователю электронное письмо с просьбой подтвердить существование адреса электронной почты, занесенного при регистрации.

Пример привязки слушателя `UserAttemptListener` к событию `Attempting`:

```
class EventServiceProvider extends ServiceProvider {
    protected $listen = [
        . . .
        \Illuminate\Auth\Events\Attempting::class => [
            \App\Listeners\UserAttemptListener::class,
        ],
    ];
    . . .
}
```

Для привязки слушателей-классов к событиям также можно использовать метод `listen()` фасада `Illuminate\Support\Facades\Event`:

```
listen(<путь к классу события>|<массив путей к классам событий>,
       <путь к классу слушателя>)
```

Вызов этого метода можно поместить, например, в метод `boot()` провайдера `EventServiceProvider`:

```
use Illuminate\Support\Facades\Event;
. . .
public function boot() {
    parent::boot();
    Event::listen(\Illuminate\Auth\Events\Attempting::class,
                  \App\Listeners\UserAttemptListener::class);
}
```

Можно привязать один и тот же слушатель сразу к нескольким событиям, указав в вызове метода `listen()` *массив с путями* к классам этих событий:

```
Event::listen([\Illuminate\Auth\Events\Attempting::class,
               \Illuminate\Auth\Events\Login::class,
               \Illuminate\Auth\Events\Failed::class],
              \App\Listeners\UserAttemptListener::class);
```

Метод `forget(<путь к классу события>)` фасада `Event` удаляет все обработчики, привязанные к событию, *путь к классу* которого был указан:

```
Event::forget(\Illuminate\Auth\Events\Login::class);
```

Метод `hasListeners(<путь к классу события>)` фасада `Event` возвращает `true`, если к событию, *путь к классу* которого указан, был привязан хотя бы один обработчик, и `false` — в противном случае.

22.1.1.3. Автоматическая привязка слушателей-классов к событиям

Если в составе проекта присутствует много слушателей событий, выполнение их явной привязки к событиям в массиве из свойства `listen` провайдера `EventServiceProvider` может оказаться трудоемким. В таком случае можно активизировать автоматическую привязку слушателей.

При активной автоматической привязке фреймворк во время инициализации сайта просматривает все слушатели-классы, хранящиеся в определенной папке (по умолчанию: app\Listeners), и ищет в них методы `handle()`. Очередной слушатель привязывается к тому событию, которое указано в качестве типа единственного параметра его метода `handle()` (например, если в качестве типа указано событие `Attempting`, слушатель будет привязан к этому событию).

Чтобы активизировать автоматическую привязку слушателей, достаточно в общедоступном методе `shouldDiscoverEvents()` провайдера `EventServiceProvider` изменить возвращаемый результат с `false` на `true`:

```
class EventServiceProvider extends ServiceProvider {
    . . .
    public function shouldDiscoverEvents() {
        return true;
    }
}
```

Если модули с классами слушателей хранятся в папке, отличной от используемой по умолчанию (app\Listeners), или сразу в нескольких папках, нужно дать Laravel указание, где их искать. Для этого следует объявить в том же провайдере защищенный метод `discoverEventsWithin()`, не принимающий параметров и возвращающий массив с полными путями к этим папкам. Пример:

```
class EventServiceProvider extends ServiceProvider {
    . . .
    protected function discoverEventsWithin() {
        return [app_path('Listeners'), app_path('EventHandlers')];
    }
}
```

При активизированной автоматической привязке слушателей-классов также будут работать слушатели-классы, привязанные явно.

22.1.1.4. Слушатели-функции

Если слушатель достаточно прост, его можно оформить как анонимную функцию. Привязка слушателей-функций к событиям выполняется в теле метода `boot()` провайдера `EventServiceListener` с помощью метода `listen()` фасада `Illuminate\Support\Facades\Event`:

```
listen(<путь к классу события>|<массив путей к классам событий>,
       <анонимная функция-слушатель события>)
```

Слушатель-функцию можно привязать как к одному событию, указав *путь* к его классу, так и сразу к нескольким событиям, задав *массив путей* к их классам. Сам *слушатель-функция* должен принимать в качестве параметра объект события.

Пример привязки слушателя-функции к событию `Attempting`:

```
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Storage;
class EventServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        parent::boot();

        Event::listen(\Illuminate\Auth\Events\Attempting::class,
               function ($event) {
                   $s = 'email: ' . $event->credentials['email'] .
                        ', password: ' .$event->credentials['password'];
                   Storage::disk('local')->append('attempts.log', $s);
               });
    }
}
```

Поддерживается сокращенный формат вызова этого метода: `listen(<слушатель-функция>)`. Только в этом случае у параметра задаваемого слушателя-функции нужно указать в качестве типа класс обрабатываемого события. Пример:

```
Event::listen(
       function (\Illuminate\Auth\Events\Attempting $event) {
           . . .
       });
```

22.1.1.5. Просмотр списков слушателей, привязанных к событиям-классам

Просмотреть список слушателей, классов и функций, привязанных к событиям-классам как явно, так и автоматически, можно, набрав команду:

```
php artisan event:list [--event=<путь к классу события>]
```

Список выводится в виде таблицы из двух столбцов: **Event** (путь к классу события) и **Listeners** (пути к классам слушателей и слушателям-функциям, последние предваряются словом **Closure**). По умолчанию выводятся слушатели всех событий.

Если указать командный ключ `--event`, будут выведены лишь слушатели события-класса с заданным *путем*:

```
php artisan event:list --event=Illuminate\Auth\Events\Registered
```

Как показала практика, полный путь указывать необязательно — достаточно ввести его фрагмент или одно лишь имя класса события:

```
php artisan event:list --event=Registered
```

22.1.2. Обработка событий-классов: подписчики

Подписчик (subscriber) объединяет в себе несколько слушателей, обрабатывающих разные события, и реализуется в виде класса.

Утилита artisan не умеет создавать подписчики, поэтому придется делать это вручную. Обычно класс подписчика объявляется в пространстве имен `App\Listeners` и не является ничьим подклассом. В нем должны быть объявлены следующие общедоступные методы:

- методы, собственно обрабатывающие события. Могут иметь произвольные имена и должны в качестве единственного параметра принимать объект события;
- метод `subscribe()` — выполняющий привязку событий к обрабатывающим их методам. В качестве параметра должен принимать объект подсистемы событий.

  Результатом, возвращаемым этим методом, должен быть ассоциативный массив привязок. Ключами элементов этого массива станут пути к классам событий, а значениями элементов — имена методов-обработчиков этих событий.

В листинге 22.2 показан код подписчика `App\Listeners\LoginFailedSubscriber`, обрабатывающий события `Illuminate\Auth\Events\Login` (генерируется при успешном входе) и `Illuminate\Auth\Events\Failed` (генерируется при неуспешном входе).

Листинг 22.2. Код подписчика `App\Listeners\LoginFailedSubscriber`

```
namespace App\Listeners;
use Illuminate\Support\Facades\Storage;
class LoginFailedSubscriber {
    public function handleLogin($event) {
        Storage::disk('local')->append('attempts.log', 'Success!');
    }

    public function handleFailed($event) {
        Storage::disk('local')->append('attempts.log', 'Failed!');
    }

    public function subscribe($events) {
        return [
            \Illuminate\Auth\Events\Login::class => 'handleLogin',
```

```
            \Illuminate\Auth\Events\Failed::class => 'handleFailed'
        ];
    }
}
```

Регистрируются подписчики в массиве, присваиваемом защищенному свойству `subscribe` провайдера `EventServiceProvider`. Элементами этого массива должны быть пути к классам подписчиков. Пример:

```
class EventServiceProvider extends ServiceProvider {
    . . .
    protected $subscribe = [
        \App\Listeners\LoginFailedSubscriber::class,
    ];
    . . .
}
```

Для той же цели можно использовать метод `subscribe(<путь к классу подписчика>)` фасада `Event`, поместив его вызов, например, в метод `boot()` провайдера `EventServiceProvider`:

```
public function boot() {
    parent::boot();
    Event::subscribe(\App\Listeners\LoginFailedSubscriber::class);
}
```

22.1.3. События-классы, поддерживаемые фреймворком

22.1.3.1. События подсистемы разграничения доступа

Все рассматриваемые далее классы событий, генерируемых подсистемой разграничения доступа, объявлены в пространстве имен `Illuminate\Auth\Events`:

- `Registered` — генерируется сразу после сохранения в списке пользователей вновь зарегистрированного пользователя.

  Свойство `user` хранит нового пользователя в виде объекта модели `User`;

- `Attempting` — генерируется при попытке входа перед проверкой существования пользователя с заданными адресом электронной почты и паролем. Поддерживаются свойства:

  - `credentials` — ассоциативный массив с данными, введенными в веб-форму входа. Обычно содержит элементы `email` (адрес электронной почты, по умолчанию используемый Laravel для идентификации пользователя) и `password` (пароль в незашифрованном виде);
  - `remember` — `true`, если было активизировано запоминание пользователя, и `false` — в противном случае;
  - `guard` — имя используемого стража;

- `Validated` — генерируется сразу после успешной проверки существования зарегистрированного пользователя с заданными адресом электронной почты и паролем, но перед собственно выполнением входа от его имени. Поддерживаются свойства:
  - `user` — пользователь, выполняющий вход, в виде объекта модели `User`;
  - `guard` — имя используемого стража;
- `Login` — генерируется сразу после успешного входа. Поддерживаются:
  - `user` — пользователь, выполнивший вход, в виде объекта модели `User`;
  - `remember` — `true`, если было активизировано запоминание пользователя, и `false` — в противном случае;
  - `guard` — имя используемого стража;
- `Authenticated` — генерируется, когда текущий пользователь идентифицируется как зарегистрированный пользователь, выполнивший вход (что происходит при вызове метода `user()` фасада `Auth`, описанного в *разд. 13.9*). Поддерживаются те же свойства, что и у класса события `Validated`;
- `Logout` — генерируется сразу после успешного выхода. Поддерживаются те же свойства, что и у класса события `Validated`;
- `CurrentDeviceLogout` — генерируется после завершения пользовательской сессии на текущем устройстве. Поддерживаются те же свойства, что и у класса события `Validated`;
- `OtherDeviceLogout` — генерируется после завершения сессий, открытых текущим пользователем, на других устройствах. Поддерживаются те же свойства, что и у класса события `Validated`;
- `Failed` — генерируется при неуспешной попытке входа. Поддерживаются свойства:
  - `credentials` — ассоциативный массив с данными, введенными в веб-форму входа;
  - `user` — пользователь, успешно найденный в списке пользователей по заданным им в веб-форме данным, но не допускаемый на сайт по каким-либо причинам (например, указанный у него адрес электронной почты еще не был проверен). Если занесенные в веб-форму входа данные не соответствуют ни одному из зарегистрированных пользователей, это свойство хранит `null`;
  - `guard` — имя используемого стража;
- `Lockout` — генерируется, когда страница входа временно блокируется после нескольких безуспешных попыток войти на сайт, выполненных подряд.

  Свойство `request` хранит объект клиентского запроса, вызвавшего блокировку;
- `Verified` — генерируется сразу после успешной проверки существования адреса электронной почты, заданного пользователем при регистрации. Поддерживается то же свойство, что и у класса события `Registered`;
- `PasswordReset` — генерируется сразу после успешного сброса пароля.

  Свойство `user` хранит пользователя, сбросившего свой пароль, в виде объекта модели `User`.

22.1.3.2. События других подсистем

- `Illuminate\Routing\Events\RouteMatched` — генерируется при совпадении пути, извлеченного из поступившего клиентского запроса, с одним из маршрутов. Поддерживаются свойства:
  - `request` — объект клиентского запроса;
  - `route` — объект класса `Illuminate\Routing\Route`, представляющий совпавший маршрут;
- `Illuminate\Foundation\Http\Events\RequestHandled` — генерируется после успешной обработки клиентского запроса и формирования серверного ответа. Поддерживаются свойства:
  - `request` — объект клиентского запроса;
  - `response` — объект серверного ответа.

22.1.4. Создание и использование своих событий-классов

22.1.4.1. Создание событий-классов

Создать новый класс события можно подачей команды:

```
php artisan make:event <имя класса события>
```

Класс события объявляется в пространстве имен `App\Events` (соответствующая папка создается автоматически) и не является ничьим подклассом. Изначально он включает три трейта:

- `Illuminate\Foundation\Events\Dispatchable` — содержит статические методы: `dispatch()`, `dispatchIf()` и `dispatchUnless()`, описываемые далее. Если эти методы не используются, трейт можно удалить, сэкономив немного системных ресурсов;
- `Illuminate\Broadcasting\InteractsWithSockets` — используется при написании событий, транслируемых по каналам вещания (будут описаны в *главе 31*). Если событие не будет транслироваться, трейт можно удалить из класса;
- `Illuminate\Queue\SerializesModels` — обеспечивает сериализацию объектов событий для записи в очередь для последующей обработки в отложенных обработчиках (будут описаны в *главе 25*). Если создаваемое событие не будет обрабатываться в отложенных обработчиках, трейт можно удалить.

В классе события изначально объявлены два общедоступных метода:

- конструктор — должен принимать в качестве параметров значения, которые необходимо сохранить в объекте события, и выполнять их сохранение. Свойства, в которых будут записываться сохраняемые значения, следует объявить в классе события вручную как общедоступные;
- `broadcastOn()` — используется при пересылке события посредством вещания (будет рассмотрено в *главе 31*).

В листинге 22.3 показан код события App\Events\BbUpdated, генерируемого при сохранении исправленного объявления. Класс этого события поддерживает свойства bb (исправленное объявление) и user (пользователь, исправивший объявление).

Листинг 22.3. Код события App\Events\BbUpdated

```
namespace App\Events;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;
class BbUpdated {
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $bb;
    public $user;

    public function __construct($bb, $user) {
        $this->bb = $bb;
        $this->user = $user;
    }

    public function broadcastOn() {
        return new PrivateChannel('channel-name');
    }
}
```

К созданному таким образом событию-классу можно привязать обработчик — слушатель или подписчик. Делается эти теми же способами, что были описаны в *разд. 22.1.1* и *22.1.2*.

22.1.4.2. Создание событий-классов и их слушателей

Также есть возможность сразу создать произвольное количество событий-классов и обрабатывающих их слушателей-классов набором одной команды. Для этого необходимо выполнить два действия:

1. В массив из свойства listen провайдера EventServiceProvider (см. *разд. 22.1.1.2*) добавить события, которые требуется создать, вместе с привязанными к ним слушателями-классами.
2. Исполнить команду:

```
php artisan event:generate
```

В результате будут созданы события-классы, присутствующие в массиве из свойства listen, но еще не существующие, и привязанные к ним, но также еще не существующие слушатели-классы.

22.1.4.3. Генерирование своих событий

Сгенерировать созданное событие-класс можно следующими способами:

- вызвать метод `dispatch(<объект события>)` фасада `Event`:

```
use Illuminate\Support\Facades\Event;
use App\Events\BbUpdated;
. . .
$bb->save();
Event::dispatch(new BbUpdated($bb, request()->user()));
```

- вызвать функцию-хелпер `event()`, имеющую тот же формат вызова, что и метод `dispatch()`, и выполняющуюся чуть медленнее:

```
event(new BbUpdated($bb, request()->user()));
```

- если класс события содержит трейт `Illuminate\Foundation\Events\Dispatchable` (впрочем, любое событие содержит его изначально) — вызвать у класса события один из следующих статических методов:
  - `dispatch()` — просто генерирует событие, при создании его объекта передавая конструктору заданные *параметры*:

    `dispatch(<параметр 1>, <параметр 2>, ... <параметр n>)`

    Пример:

    ```
    BbUpdated::dispatch($bb, $request->user());
    ```

  - `dispatchIf()` — аналогичен методу `dispatch()`, только генерирует событие, если заданное *условие* истинно (в результате вычисления дает `true`):

    `dispatchIf(<условие>, <парам. 1>, <парам. 2>, ... <парам. n>)`

    Пример:

    ```
    // Генерируем событие только в случае успешного сохранения
    // объявления
    BbUpdated::dispatchIf($bb->save(), $bb, $request->user());
    ```

  - `dispatchUnless()` — аналогичен методу `dispatch()`, только генерирует событие, если заданное *условие* ложно (в результате вычисления дает `false`). Формат вызова такой же, как и у метода `dispatchIf()`.

22.2. События-строки

Событие-строка представляется не классом, а уникальным именем, записанным в виде строки. Обработчику такого события при его генерировании можно передать произвольное количество параметров.

Поддержка событий-строк осталась в Laravel со времен старых версий. В настоящее время их можно встретить в ранее написанном коде. Использовать их при программировании новых сайтов рекомендуется лишь в наиболее простых случаях (например, если в составе сайта создается лишь одно событие).

22.2.1. Привязка обработчиков к событиям-строкам

Специально создавать события-строки не нужно — достаточно лишь привязать к ним обработчики, указав в выражениях, выполняющих привязку, нужные имена событий. Сами выражения привязки пишутся в теле метода `boot()` провайдера `EventServiceProvider` и включают метод `listen()` фасада `Event`, вызываемый в формате:

```
listen(<имя события>|<шаблон имен событий>,
       <анонимная функция-обработчик события>)
```

Анонимная функция может принимать параметры, указанные при генерировании события и содержащие различные сведения о нем. Пример:

```
use Illuminate\Support\Facades\Event;
. . .
Event::listen('bb.updated', function ($bb, $user) {
    $s = 'title: ' . $bb->title . ', username: ' . $user->name;
    Storage::disk('local')->append('bbs.log', $s);
});
```

Если один и тот же обработчик должен обрабатывать сразу несколько событий с похожими именами, первым параметром методу `listen()` можно передать *шаблон*, с которым должны совпадать имена обрабатываемых событий. *Шаблон* записывается в виде строки и может включать литерал `*`, обозначающий произвольное количество любых символов. Указываемая *анонимная функция* в этом случае должна принимать два параметра: имя обрабатываемого события в виде строки и индексированный массив параметров, переданных обработчику при генерировании события.

Пример обработчика, обрабатывающего события: `bb.stored`, `bb.created`, `bb.destroying` и т. п.:

```
Event::listen('bb.*', function ($eventName, $params) {
    $s = 'event: ' . $eventName . ', title: ' . $params[0]->title .
         ', username: ' . $params[1]->name;
    Storage::disk('local')->append('bbs.log', $s);
});
```

Для проверки, был ли к указанному событию-строке привязан обработчик, можно использовать метод `hasListeners()` (см. *разд. 22.1.1.2*), указав в качестве параметра имя события.

Метод `hasWildcardListeners(<шаблон имен событий>)` фасада `Event` возвращает `true`, если к событиям, имена которых совпадают с заданным *шаблоном*, был привязан хотя бы один обработчик, и `false` — в противном случае.

22.2.2. Генерирование событий-строк

Для генерирования событий-строк применяются инструменты, описанные в *разд. 22.1.4.3*, только с отличающимися форматами вызова:

- метод `dispatch()` фасада `Event` — вызывается в формате:

  ```
  dispatch(<имя события>[, <массив с параметрами>=[]])
  ```

Массив с параметрами, передаваемыми обработчикам события, должен быть индексированным. Пример:

```
Event::dispatch('bbs.updated', [$bb, request()->user()]);
```

Если обработчик события принимает всего один параметр, его значение можно указать непосредственно, не заключая в массив:

```
Event::dispatch('bbs.destroying', $bb);
```

- функция-хелпер `event()` — вызывается в том же формате, что и метод `dispatch()`.

22.3. События моделей

События моделей — это те же самые события-строки, только привязка к ним обработчиков выполняется непосредственно в классах моделей.

22.3.1. Обработка событий моделей

22.3.1.1. Обработка событий моделей посредством слушателей-функций

Проще всего обрабатывать события моделей посредством слушателей-функций. Их привязка выполняется в теле статического защищенного метода `booted()`, не принимающего параметров и объявляемого в классе модели.

Для привязки слушателя к событию применяется статический метод, который вызывается у класса модели и имеет имя, совпадающее с именем обрабатываемого события (имена событий моделей будут приведены далее). В качестве единственного параметра этот метод принимает анонимную функцию, реализующую слушатель и принимающую в качестве параметра объект модели.

Пример привязки обработчика к событию `updated`, генерируемому при сохранении исправляемой записи:

```
use Illuminate\Support\Facades\Storage;
class Bb extends Model {
    . . .
    protected static function booted() {
        static::updated(function ($bb) {
            $s = 'title: ' . $bb->title;
            Storage::disk('local')->append('bbs.log', $s);
        });
    }
}
```

22.3.1.2. Связывание событий моделей с событиями-классами

Также можно связать нужное событие модели с подходящим событием-классом. После чего при генерировании моделью этого события фреймворк автоматически сгенерирует связанное с ним событие-класс, для обработки которого можно использовать любые

способы, описанные в *разд. 22.1*. Это позволяет обрабатывать все события сайта единым образом.

Для связывания события модели с событием-классом надо выполнить следующие шаги:

1. Удостовериться, что конструктор события-класса принимает всего один параметр (по крайней мере обязательный) — объект модели. Пример класса с таким конструктором:

```
class BbUpdated {
    public $bb;

    public function __construct($bb) {
        $this->bb = $bb;
    }
    . . .
}
```

2. Объявить в классе модели защищенное свойство `dispatchesEvents` и присвоить ему ассоциативный массив связок. Ключами элементов такого массива должны быть имена событий модели, а значениями элементов — пути к связанным с ними событиям-классам. Пример:

```
class Bb extends Model {
    . . .
    protected $dispatchesEvents = [
        'updated' => \App\Events\BbUpdated::class,
    ];
    . . .
}
```

22.3.1.3. Использование обозревателей

Обозреватель (observer) — это аналог подписчика (см. *разд. 22.1.2*), только для обработки событий моделей.

Новый класс обозревателя создается командой:

```
php artisan make:observer <имя класса обозревателя> ↳
[--model=<имя класса модели>]
```

Класс обозревателя объявляется в пространстве имен `App\Observers` (соответствующая папка создается автоматически) и не является ничьим подклассом.

По умолчанию создается «пустой» класс обозревателя. В нем объявляются методы, которые станут обрабатывать события модели. Имя такого метода должно совпадать с обрабатываемым им событием модели (например, для обработки события `created` нужно объявить метод `created()`), а принимать этот метод должен единственный параметр — объект модели.

Если был указан командный ключ `--model`, будет создан класс обозревателя, предназначенный для обработки событий модели, класс которой имеет заданное *имя*. Такой класс изначально будет содержать методы: `created()`, `updated()`, `deleted()`, `restored()` и `forceDeleted()`, обрабатывающие одноименные события.

Можно дать Laravel указание выполнить обработчик возникшего события только после того, как транзакция, внутри которой выполняются операции с базой данных, будет подтверждена. Сделать это можно, объявив в классе обозревателя общедоступное свойство `afterCommit` и присвоив ему значение `true`.

В листинге 22.4 показан код обозревателя `App\Observers\BbObserver`, который обрабатывает событие `updated` модели `Bb`.

Листинг 22.4. Код обозревателя `App\Observers\BbObserver`

```
namespace App\Observers;
use Illuminate\Support\Facades\Storage;
use App\Models\Bb;
class BbObserver {
    public $afterCommit = true;

    public function created(Bb $bb) { }

    public function updated(Bb $bb) {
        $s = 'title: ' . $bb->title. ', price: ' . $bb->price;
        Storage::disk('local')->append('bbs.log', $s);
    }

    public function deleted(Bb $bb) { }

    public function restored(Bb $bb) { }

    public function forceDeleted(Bb $bb) { }
}
```

Далее остается связать обозреватель с моделью. Связывание выполняется в теле метода `boot()` провайдера `EventServiceProvider` вызовом у модели статического метода `observe(<путь к классу обозревателя>)`. Пример:

```
use App\Models\Bb;
use App\Observers\BbObserver;
class EventServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Bb::observe(BbObserver::class);
    }
}
```

22.3.2. Список событий моделей

Все события моделей передают обработчикам единственный параметр — объект модели, обрабатываемый в текущий момент. Список событий моделей приведен в табл. 22.1.

Таблица 22.1. *События моделей, поддерживаемые Laravel*

Имя	Генерируется
`creating`	Перед сохранением новой записи
`created`	После сохранения новой записи
`retrieved`	После выборки существующей записи из таблицы
`updating`	Перед сохранением существующей записи
`updated`	После сохранения существующей записи
`saving`	Перед сохранением новой или существующей записи, до события `creating` или `updating`
`saved`	После сохранения новой или существующей записи, после события `created` или `updated`
`deleting`	Перед удалением записи
`deleted`	После удаления записи
`restoring`	Перед восстановлением записи, подвергшейся «мягкому» удалению
`restored`	После восстановления записи, подвергшейся «мягкому» удалению
`forceDeleted`[1]	После полного удаления записи, если модель поддерживает «мягкое» удаление

Если из обработчика события: `creating`, `updating`, `saving`, `deleting` или `restoring` вернуть в качестве результата `false`, соответствующая операция будет отменена. Например, так можно отменить создание объявления:

```
static::creating(function ($bb) {
    . . .
    return false;
});
```

22.3.3. Временное отключение событий в моделях

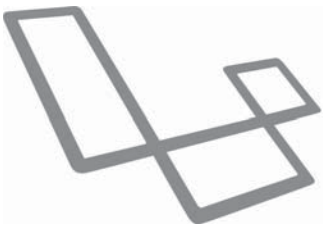
Иногда может понадобиться временно отключить генерирование событий в какой-либо модели. Для этого достаточно вызвать у этой модели статический метод `withoutEvents(<`*`анонимная функция`*`>)` и записать действия, которые следует выполнить без генерирования событий, в заданной *анонимной функции*. Последняя не должна принимать параметры, но может возвращать результат. Пример:

```
$rubric = Rubric::withoutEvents(function () use ($rubric_id) {
    return Rubric::find($rubric_id);
});
```

Если требуется лишь сохранить запись, не генерируя событий, можно воспользоваться методом `saveQuietly()` модели, имеющим тот же формат вызова, что и метод `save()`:

```
$bb->saveQuietly();
```

[1] Это событие можно обрабатывать только в обозревателях.



ГЛАВА 23

Отправка электронной почты

23.1. Настройки подсистемы отправки электронной почты

Все основные настройки этой подсистемы хранятся в модуле config\mail.php:

- `mailers` — ассоциативный массив почтовых служб, посредством которых будут отправляться письма. Ключи элементов этого массива задают имена служб, а значения элементов представляют собой вложенные ассоциативные массивы, содержащие следующие настройки этих служб:
  - `transport` — тип службы. Поддерживаются следующие типы:
    - `smtp` — SMTP-сервер;
    - `ses` — служба Amazon SES. Для ее использования следует установить дополнительную библиотеку, набрав команду:

      ```
      composer require aws/aws-sdk-php
      ```

    - `mailgun` — служба Mailgun. Для ее использования следует установить дополнительную библиотеку, набрав команду:

      ```
      composer require symfony/mailgun-mailer symfony/http-client
      ```

    - `postmark` — служба Postmark. Для ее использования следует установить дополнительные библиотеки, набрав команды:

      ```
      composer require symfony/postmark-mailer symfony/http-client
      ```

    - `sendmail` — программа Sendmail или аналогичная ей;
    - `log` — файл журнала. Используется только при отладке;
    - `array` — массив, хранящийся в свойстве объекта соответствующей службы. Используется только при отладке;
    - `failover` — агрегатор, объединяющий несколько почтовых служб и обеспечивающий бесперебойную рассылку писем. Если первая почтовая служба из указанных в списке этого агрегатора перестанет работать, письма будут рассылаться с помощью второй службы и т. д.

Следующие настройки используются только службой `smtp`:

- `host` — интернет-адрес SMTP-сервера. Значение берется из локальной настройки `MAIL_HOST`, имеющей значение `mailhog`. Значение по умолчанию: `smtp.mailgun.org`;
- `port` — номер TCP-порта, через который работает SMTP-сервер, в виде целого числа. Значение берется из локальной настройки `MAIL_PORT`, имеющей значение `1025`. По умолчанию: `587`;
- `encryption` — обозначение протокола шифрования, используемого для доступа к SMTP-серверу. Поддерживаются значения: `'ssl'` — будет использован протокол SSL (Secure Sockets Layer, уровень защищенных сокетов), `'tls'` — протокол TLS (Transport Layer Security, протокол защиты транспортного уровня) и `null` (отсутствие шифрования). Значение берется из локальной настройки `MAIL_ENCRYPTION`, имеющей значение `null`. По умолчанию: `'tls'`;
- `username` — имя пользователя для доступа к SMTP-серверу. Значение берется из локальной настройки `MAIL_USERNAME`, имеющей значение `null`;
- `password` — пароль для доступа к SMTP-серверу. Значение берется из локальной настройки `MAIL_PASSWORD`, имеющей значение `null`;
- `timeout` — время ожидания подключения к SMTP-серверу в виде целого числа в секундах. Если указать `null`, время ожидания будет неограниченным. По умолчанию: `null`.

Пример указания настроек SMTP-сервера в файле .env:

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mail.ru
MAIL_PORT=465
MAIL_USERNAME=support@bboard.ru
MAIL_PASSWORD=1234567890
MAIL_ENCRYPTION=ssl
```

Следующая настройка используется только службой `sendmail`:

- `path` — команда для запуска Sendmail в виде строки. Значение берется из локальной настройки `MAIL_SENDMAIL_PATH`, изначально отсутствующей. Значение по умолчанию: `'/usr/sbin/sendmail -bs'`.

Следующая настройка используется только службой `log`:

- `channel` — имя канала журналирования, используемого для вывода писем (подсистема журналирования будет описана в *главе 34*). Значение берется из локальной настройки `MAIL_LOG_CHANNEL`, изначально отсутствующей.

Следующая настройка используется только службой `failover`:

- `mailers` — массив имен почтовых служб, используемых агрегатором (по умолчанию содержит службы `smtp` и `log`).

Изначально присутствуют службы: `smtp`, `ses`, `mailgun`, `postmark`, `sendmail`, `log`, `array` и `failover`;

- `default` — служба по умолчанию, используемая для отправки писем, если служба не была указана явно. Берет значение из локальной настройки `MAIL_MAILER`. По умолчанию: `smtp`;

- `from` — имя и адрес электронной почты отправителя, заносимый в отправляемые письма, если имя и адрес не были заданы явно. Содержит ассоциативный массив с двумя настройками:
  - `address` — адрес электронной почты отправителя. Значение берется из локальной настройки `MAIL_FROM_ADDRESS`. По умолчанию: `hello@example.com`;
  - `name` — имя отправителя. Значение берется из локальной настройки `MAIL_FROM_NAME`, в свою очередь, получающей значение из локальной настройки `APP_NAME`. По умолчанию: `'Example'`.

  Пример указания имени и адреса отправителя в файле .env:

```
MAIL_FROM_ADDRESS=support@bboard.ru
MAIL_FROM_NAME='Служба поддержки BBoard.ru'
```

- `reply_to` — имя и адрес электронной почты получателя ответов. Значение указывается в том же формате, что и у настройки `from`. Изначально отсутствует;
- `markdown` — настройки встроенного шаблонизатора Markdown. Содержит ассоциативный массив с двумя настройками (более подробно они будут описаны в этой главе далее):
  - `theme` — имя темы оформления писем (по умолчанию: `default`);
  - `paths` — массив полных путей к шаблонам Markdown. По умолчанию он содержит единственный элемент — путь к папке resources\views\vendor\mail, изначально не существующей.

Настройки, необходимые для взаимодействия со службами Amazon SES, Mailgun и Postmark, хранятся в модуле config\services.php. Все они содержат ассоциативные массивы с настройками подключения к соответствующей службе:

- `ses` — настройки службы Amazon SES:
  - `key` — ключ доступа. Значение берется из локальной настройки `AWS_ACCESS_KEY_ID`, присутствующей в файле .env, но изначально «пустой»;
  - `secret` — секретный ключ. Значение берется из локальной настройки `AWS_SECRET_ACCESS_KEY`, присутствующей в файле .env, но изначально «пустой»;
  - `region` — обозначение региона. Значение берется из локальной настройки `AWS_DEFAULT_REGION`. По умолчанию: `us-east-1`;
  - `token` — электронный жетон для временного доступа. Изначально отсутствует;
  - `options` — ассоциативный массив с дополнительными параметрами отправки писем (описываются в документации по службе Amazon SES);
- `mailgun` — настройки службы Mailgun:
  - `domain` — домен службы. Значение берется из локальной настройки `MAILGUN_DOMAIN`, изначально отсутствующей;
  - `secret` — секретный ключ. Значение берется из локальной настройки `MAILGUN_SECRET`, изначально отсутствующей;
  - `endpoint` — имя используемой точки контроля. Значение берется из локальной настройки `MAILGUN_ENDPOINT`, изначально отсутствующей. По умолчанию: `'api.mailgun.net'`;

- `scheme` — обозначение протокола, используемого для взаимодействия со службой: `'http'` или `'https'` (по умолчанию: `'https'`);

❒ `postmark` — настройки службы Postmark:

- `token` — электронный жетон службы. Значение берется из локальной настройки `POSTMARK_TOKEN`, изначально отсутствующей;
- `message_stream_id` — идентификатор потока, используемого для отправки писем. Изначально отсутствует.

23.2. Создание электронных писем

Для создания электронного письма в Laravel необходимо написать:

❒ класс письма;

❒ шаблон для письма в текстовом формате;

❒ шаблон для письма в формате HTML.

Если какое-либо письмо планируется отправлять только в текстовом формате, шаблон в формате HTML создавать не нужно, и наоборот.

23.2.1. Создание классов электронных писем

Для создания нового класса письма надо выполнить команду:

```
php artisan make:mail <имя класса письма> ↳
[--markdown=<путь к шаблону Markdown>] [--force]
```

По умолчанию создается один лишь класс письма без шаблонов — их придется делать самостоятельно.

Поддерживаются командные ключи:

❒ `--markdown` — создает класс, изначально предназначенный для генерирования писем на основе шаблонов, написанных на языке Markdown, и шаблон такого письма, сохраняемый по заданному *пути*, который отсчитывается от папки resources\views;

❒ `--force` — принудительно создает класс письма, даже если одноименный модуль уже существует.

Класс письма объявляется в пространстве имен `App\Mail` (соответствующая папка создается автоматически) и является производным от класса `Illuminate\Mail\Mailable`. Изначально он включает трейты `Illuminate\Bus\Queueable` и `Illuminate\Queue\SerializesModels`, которые реализуют функциональность отложенных писем (см. *главу 25*) и могут быть удалены, если создается обычное, не отложенное письмо.

В классе письма должны присутствовать два метода:

❒ конструктор — изначально «пустой». Может использоваться для получения каких-либо значений, в том числе и посредством внедрения зависимостей;

❒ `build()` — собственно генерирует электронное письмо на основе заданных шаблонов. Не должен принимать параметров и должен возвращать в качестве результата текущий объект.

В листинге 23.1 показан код класса простого тестового письма `App\Mail\SimpleMail`, отправляемого в текстовом формате и приветствующего пользователя, имя которого было получено через параметр конструктора.

Листинг 23.1. Код класса письма `App\Mail\SimpleMail`

```
namespace App\Mail;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
class SimpleMail extends Mailable {
    use Queueable, SerializesModels;

    private $username;

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

    public function build() {
        return $this->subject('Привет, ' . $this->username . '!')
                    ->text('mails.simple-text',
                           ['name' => $this->username]);
    }
}
```

В листинге 23.2 показан код шаблона resources\views\mails\simple-text.blade.php, создающего это письмо.

Листинг 23.2. Код шаблона resources\views\mails\simple-text.blade.php

```
Привет, {{ $name }}!

Как дела?
```

23.2.2. Генерирование электронных писем

Код, генерирующий письмо, записывается в теле метода `build()` класса письма. Генерирование осуществляется вызовами у текущего объекта письма особых методов, унаследованных от суперкласса. Все они в качестве результата возвращают текущий объект, что позволяет записывать их вызовы цепочкой. Вот эти методы:

- `subject(<`*тема*`>)` — задает *тему* текущего письма.

  Если тема не была задана явно, в ее качестве будет использовано имя класса письма;
- `view(<`*путь к шаблону*`>[, <`*контекст шаблона*`>=[]])` — задает *путь к шаблону*, на основе которого будет генерироваться письмо в формате HTML, и *контекст* этого шаблона. *Путь к шаблону* отсчитывается относительно папки resources\views, как и шаблоны

веб-страниц, и записывается по тем же правилам (см. *главу 11*). *Контекст шаблона* оформляется в виде ассоциативного массива. Пример:

```
public function build() {
    return $this->subject('Тестовое письмо в формате HTML')
                ->view('mails.simple-html',
                       ['name' => $this->username]);
}
```

- `text()` — задает путь к шаблону, на основе которого будет генерироваться письмо в текстовом формате, и контекст этого шаблона. Формат вызова такой же, как и у метода `view()`. Пример использования можно увидеть в листинге 23.1.

  Можно совместить вызовы методов `view()` и `text()` — тогда будет сгенерировано составное письмо из двух частей двух форматов: HTML и текстового:

  ```
  return $this->view('mails.simple-html', ['name' => $this->username])
              ->text('mails.simple-text');
  ```

  Следует обратить внимание, что оба шаблона используют один и тот же контекст, поэтому задавать контекст шаблона в вызове второго метода не нужно;

- `with()` — добавляет в контекст шаблона текущего письма новые переменные. Поддерживает два формата вызова:

  ```
  with(<имя переменной>, <значение переменной>)
  with(<ассоциативный массив с добавляемыми переменными>)
  ```

  Первый формат добавляет одну переменную с указанными *именем* и *значением*:

  ```
  return $this->view('mails.simple-html')->text('mails.simple-text')
              ->with('name', $this->username);
  ```

  Второй формат добавляет все переменные, содержащиеся в указанном *ассоциативном массиве*. Ключи элементов этого массива зададут имена переменных, а значения элементов — значения этих переменных. Пример:

  ```
  return $this->view('mails.simple-html')
         ->with(['name' => $this->username, 'email' => $this->email]);
  ```

- `html(<HTML-код>)` — задает в качестве содержимого текущего письма указанный *HTML-код*:

  ```
  return $this->subject('Письмо в формате HTML')
              ->html('<h1>Привет, пользователь!</h1>');
  ```

- `from(<адрес отправителя>[, <имя отправителя>=null])` — указывает у текущего письма *адрес* электронной почты и необязательное *имя* отправителя вместо записанных в настройке `mail.from`;

- `replyTo()` — указывает у текущего письма адрес электронной почты и необязательное имя получателя ответов вместо записанных в настройке `mail.reply_to`. Формат вызова такой же, как и у метода `from()`;

- `priority([<приоритет>=3])` — указывает *приоритет* текущего письма в виде целого числа от 1 (наивысший) до 5 (наинизший);

- `attach()` — добавляет в текущее письмо файл с заданным *путем* в виде вложения:

  ```
  attach(<путь к файлу>[, <ассоциативный массив с параметрами>=[]])
  ```

Пример:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attach('c:/site/mail/attaches/license.doc');
```

В *ассоциативном массиве* ключи элементов зададут имена параметров, а значения элементов — значения этих параметров. Поддерживаются два параметра:

- `as` — новое имя вкладываемого файла (если не указан, файл будет вложен под своим изначальным именем);
- `mime` — MIME-тип вкладываемого файла. Указывается, если файл сохранен с нехарактерным для своего типа расширением (например, если файл формата DOC сохранен с расширением rtf).

Пример:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attach('c:/site/mail/attaches/license.doc', [
                        'as' => 'file.rtf',
                        'mime' => 'application/msword'
                    ]);
```

- `attachFromStorage()` — добавляет в текущее письмо файл с заданным *путем*, находящийся в хранилище Laravel по умолчанию, под указанным *именем* в виде вложения:

```
attachFromStorage(<путь к файлу>[, <имя вкладываемого файла>=null[,
                  <ассоциативный массив с параметрами>=[]]])
```

Пример:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attachFromStorage('/docs/table.xls', 'price.xls');
```

Если *имя файла* не указано, файл будет вложен в письмо под своим изначальным именем:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attachFromStorage('/docs/offers.xls');
```

В *ассоциативном массиве* можно указать параметр `mime`;

- `attachFromStorageDisk()` — аналогичен `attachFromStorage()`, только позволяет вложить в письмо файл из произвольного хранилища, задав его *имя*:

```
attachFromStorageDisk(<имя хранилища>, <путь к файлу>[,
                      <имя вкладываемого файла>=null[,
                      <ассоциативный массив с параметрами>=[]]])
```

Пример:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attachFromStorageDisk('local', 'attempts.log');
```

- `attachData()` — добавляет в текущее письмо произвольные *данные* в виде файлового вложения с указанным *именем*:

```
attachData(<данные>, <имя вкладываемого файла>[,
                     <ассоциативный массив с параметрами>=[]])
```

Пример:

```
return $this->view('mails.simple-html', ['name' => $this->username])
            ->attachData('7664856974', 'your-password.txt');
```

В *ассоциативном массиве* можно указать параметр `mime`.

23.2.3. Написание шаблонов электронных писем

Шаблоны электронных писем, как текстовые, так и в формате HTML, пишутся по тем же правилам, что и шаблоны веб-страниц (см. *главу 11*).

Выводимые в них значения можно извлечь:

- из явно созданных в контексте шаблона переменных:

```
return $this->text('mails.simple-text', ['name' => $this->username]);
. . .
Привет, {{ $name }}!
```

- из общедоступных свойств объекта письма, которые добавляются в контекст шаблона автоматически:

```
class BbMail extends Mailable {
    . . .
    public $bb;

    public function __construct($bb) {
        $this->bb = $bb;
    }
    . . .
}
. . .
<p>Товар: {{ $bb->title }}</p>
<p>Цена: {{ $bb->price }}</p>
```

Если письмо генерируется в формате HTML, в контекст шаблона также добавляется переменная `message`, хранящая объект содержимого письма (не путать с объектом собственно письма). Вызывая у этого объекта следующие методы, можно вставлять в письмо графические изображения:

- `embed(<путь к файлу>)` — вставляет в письмо изображение из файла с заданным *путем*:

```
<p><img src="{{ $message->embed('c:/site/mail/logo.jpg') }}"></p>
```

- `embedData()` — вставляет в письмо изображение, сформированное из указанных *данных*, которые оформляются в виде файла с заданным *именем*:

```
embedData(<данные>, <имя вкладываемого файла>[, <MIME-тип>=null])
```

Пример:

```
<p><img src="{{ $message->embedData($user->avatar, 'av.gif') }}"></p>
```

MIME-тип следует указывать только тогда, когда расширение имени файла не характерно для его формата (например, если данные в формате JPEG сохраняются в файле с расширением pic).

23.2.4. Написание электронных писем на языке Markdown

Markdown — облегченный язык разметки, позволяющий писать удобочитаемый и легкий для правки текст с минимумом служебных символов. Текст, написанный на Markdown, может быть легко преобразован в обычный текст или HTML.

Laravel позволяет писать на Markdown шаблоны электронных писем. На основе такого шаблона генерируются текстовая и HTML-части, впоследствии объединяемые в составное электронное письмо.

> ***Документацию по Markdown...***
>
> ...можно найти по интернет-адресам: **https://guides.hexlet.io/markdown/** (краткое русскоязычное руководство) и **https://daringfireball.net/projects/markdown/** (полная англоязычная инструкция).

23.2.4.1. Классы писем, написанных на Markdown

Класс письма, чей шаблон будет написан на языке Markdown, и сам Markdown-шаблон этого письма можно создать, набрав команду `make:mail` утилиты artisan (см. *разд. 23.2.1*) с ключом `--markdown`. Класс такого письма ничем не отличается от класса обычного письма, записываемого в виде обычного текста или HTML.

Для указания шаблона в коде метода `build()` класса такого письма следует использовать метод `markdown()`, формат вызова которого такой же, как и у методов `view()` и `text()` (см. *разд. 23.2.2*). Пример:

```
class UrgentMail extends Mailable {
    . . .
    public function build() {
        return $this->markdown('mails.urgent-markdown',
                               ['data' => $this->data]);
    }
}
```

23.2.4.2. Написание шаблонов писем на Markdown

Весь код Markdown-шаблона письма должен помещаться в компоненте `mail:message`, не принимающем параметров (о компонентах рассказывалось в *разд. 11.7*).

В листинге 23.3 представлен код Markdown-шаблона простого письма, содержащего адресное приветствие, приглашение посетить сайт и гиперссылку, ведущую на сайт и имеющую вид кнопки.

Листинг 23.3. Код простого Markdown-шаблона

```
@component('mail::message')
## Здравствуйте, {{ $name }}

Посетите, пожалуйста, наш сайт!
```

```
@component('mail::button', ['url' => 'http://www.bboard.ru/'])
Посетить сайт
@endcomponent

Спасибо,<br>
{{ config('app.name') }}
@endcomponent
```

Фреймворк предоставляет три компонента, которые можно использовать в Markdown-шаблонах:

- `mail:button` — гиперссылка в виде кнопки. Текст гиперссылки указывается непосредственно в компоненте. Поддерживает два параметра:
  - `url` — интернет-адрес;
  - `color` — цвет в виде строки: `'primary'` (темно-серый), `'success'` (зеленый) или `'error'` (красный). Если параметр не указан, кнопка будет иметь темно-серый цвет (как если бы было указано значение `'primary'`).

  Пример использования кнопки можно увидеть в листинге 23.3;
- `mail:panel` — прямоугольная панель с текстом, светло-серым фоном и тонкой темно-серой вертикальной линией вдоль левого края. Содержимое панели записывается непосредственно в компоненте. Пример:

  ```
  @component('mail::panel')
  Не забудьте зарегистрироваться на нашем сайте!
  @endcomponent
  ```

- `mail:table` — таблица. Вот пример, иллюстрирующий принципы использования компонента:

  ```
  @component('mail::table')
  | Ячейка шапки       | Ячейка шапки       | Ячейка шапки       |
  | ------------------ |:------------------:| ------------------:|
  | Ячейка содержимого | Ячейка содержимого | Ячейка содержимого |
  | Выравнивание:      | Выравнивание:      | Выравнивание:      |
  | по левому краю     |     по центру      |    по правому краю |
  @endcomponent
  ```

  Таблица выводится без рамок, только строка шапки отделяется от последующей строки очень тонкой темно-серой линией.

23.2.4.3. Управление генерированием писем, написанных на Markdown

Как говорилось ранее, весь код Markdown-шаблона записывается в компоненте `mail:message`. Три описанных в *разд. 23.2.4.2* компонента доступны для использования в шаблонах писем, и еще ряд компонентов используются шаблонизатором Markdown «за кулисами». Для преобразования этих компонентов в HTML- и текстовый форматы при генерировании писем задействуются шаблоны, входящие в составе фреймворка. Кроме того, для оформления части составного письма, записанной в формате HTML,

используется особая таблица стилей, вставляемая непосредственно в письмо и также присутствующая в составе Laravel.

Есть возможность задать для писем другое представление, более подходящее к дизайну разрабатываемого сайта. Для этого достаточно перенести упомянутые ранее шаблоны и таблицу стилей из фреймворка непосредственно в состав проекта и отредактировать нужным образом.

Чтобы перенести шаблоны и таблицу стилей, задающие представление писем на языке Markdown, из состава фреймворка в состав проекта, следует набрать команду:

```
php artisan vendor:publish --tag=laravel-mail
```

В результате будет создана папка resources\views\vendor\mail с папками html (HTML-шаблоны) и text (текстовые шаблоны). Обе папки содержат одинаковый набор из следующих шаблонов:

- message.blade.php — шаблон компонента `mail:message`, создающего само письмо. Использует компонент `mail:layout` для создания разметки письма. Содержит:
  - компонент `mail:header`, формирующий шапку письма. Шапка изначально выводит гиперссылку в виде названия проекта (берется из рабочей настройки `app.name`), ведущую на интернет-адрес хоста сайта (извлекается из рабочей настройки `app.url`);
  - содержимое письма, записанное в Markdown-шаблоне;
  - компонент `mail:subcopy`, формирующий примечание. Непонятно, зачем он нужен, поскольку никакого примечания в письме не выводится. Вероятно, это задел на будущее;
  - компонент `mail:footer`, создающий поддон письма. Поддон изначально выводит текущий год, название проекта и надпись «All rights reserved»;
- *<имя компонента без префикса mail:>*.blade.php — шаблоны остальных компонентов. Так, компонент `mail:layout` выводится с применением шаблона layout.blade.php, компонент `mail:header` — шаблона header.blade.php, а компонент `mail:button` — шаблона button.blade.php.

Можно как исправить все эти шаблоны компонентов, так и создать копию папки resources\views\vendor\mail под другим именем (например, mymail) и отредактировать находящиеся в ней копии шаблонов. Можно даже создать несколько копий этой папки, хранящих разное представление писем, и впоследствии переключаться между ними. Для переключения на другую папку с шаблонами достаточно указать полный путь к ней в единственном элементе массива из рабочей настройки `markdown.paths` (см. *разд. 23.1*):

```
'markdown' => [
    . . .
    'paths' => [
        resource_path('views/vendor/mymail'),
    ],
],
```

В папке html, вложенной в папку с шаблонами компонентов, находится вложенная папка themes. Она хранит таблицу стилей default.css, задающую оформление HTML-частей сгенерированных писем.

Исправив таблицу стилей default.css, можно изменить оформление сгенерированных писем. Также можно создать произвольное количество копий этой таблицы стилей, хранящих разное оформление (например, theme-dark.css, theme-light.css и др.), и впоследствии переключаться между ними. Указать нужную таблицу стилей для писем можно, записав ее имя без расширения в рабочую настройку `markdown.theme` (см. *разд. 23.1*):

```
'markdown' => [
    'theme' => 'theme-light',
    . . .
],
```

23.3. Отправка электронных писем

Фасад `Illuminate\Support\Facades\Mail` скрывает за собой объект службы отправки писем. Он предоставляет ряд методов, служащих для задания адресов назначения и собственно отправки электронных писем. Каждый из этих методов возвращает в качестве результата тот же объект службы отправки писем, что позволяет записывать вызовы методов цепочкой.

- `to(<`*получатель*`>|<`*массив получателей*`>)` — задает получателя или получателей письма. В качестве параметра можно задать:
  - адрес электронной почты в виде строки;
  - объект модели `User` — адрес электронной почты будет взят из поля `email`, а имя получателя — из поля `name`;
  - массив получателей — в качестве элементов которого можно указать:
    - адреса электронной почты в виде строк;
    - вложенные ассоциативные массивы с элементами `email` (адрес электронной почты) и `name` (имя получателя);
    - объекты модели `User`.

  Примеры:

```
use Illuminate\Support\Facades\Mail;
use App\Mail\MarkdownMail;
. . .
Mail::to('user@bboard.ru')->send(new MarkdownMail('user'));

$currentUser = request()->user();
Mail::to($currentUser)->send(new MarkdownMail($currentUser->name);

Mail::to([['name' => 'Вася Пупкин', 'email' => 'vasya@mail.ru'],
          'petya@gmail.com'])
    ->send(new UrgentMail());
```

Последующий вызов метода `to()` заменит адреса получателей, заданные предыдущим вызовом;

- `cc()` — задает получателя копии письма. Формат вызова и тип параметра такие же, как и у метода `to()`;
- `bcc()` — задает получателя скрытой копии письма. Формат вызова и тип параметра такие же, как и у метода `to()`. Пример:

```
Mail::to('user@bboard.ru')->cc('vasya@mail.ru')
    ->bcc('petya@gmail.com')->send(new MarkdownMail('user'));
```

- `send(<объект письма>)` — отправляет письмо, заданное в виде *объекта* нужного класса письма.

  По умолчанию отправка писем выполняется посредством службы, установленной по умолчанию (см. *разд. 23.1*);

- `mailer(<имя службы>)` — задает *имя службы*, посредством которой будет отправлено письмо. Должен был вызван непосредственно у фасада `Mail`, а все остальные методы должны вызываться у возвращенного им объекта этой службы. Пример:

```
Mail::mailer('postmark')->to('user@bboard.ru')
                        ->send(new MarkdownMail('user'));
```

Следующие три метода задают адреса получателя, отправителя и получателя ответов для всех отправляемых в дальнейшем электронных писем:

- `alwaysFrom(<адрес отправителя>[, <имя отправителя>=null])` — задает *адрес* и необязательное *имя* отправителя. Они перекрывают адрес и имя, заданные в локальной настройке `mail.from` (см. *разд. 23.1*), однако перекрываются адресом и именем, указанными в отправляемом письме;
- `alwaysReplyTo()` — задает адрес и необязательное имя получателя ответов на письма. Формат вызова такой же, как и у метода `alwaysFrom()`. Заданные в вызове адрес и имя перекрываются адресом и именем, заданными в отправляемом письме;
- `alwaysTo()` — задает *адрес* и необязательное *имя* получателя. Формат вызова такой же, как и у метода `alwaysFrom()`.

Вызовы этих методов обычно помещают в тело метода `boot()` одного из провайдеров:

```
use Illuminate\Support\Facades\Mail;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        Mail::alwaysFrom('support@bboard.ru', 'Служба поддержки');
    }
}
```

23.4. Предварительный просмотр электронных писем

Чтобы вывести содержимое письма в веб-обозревателе для просмотра, достаточно вернуть объект этого письма из контроллера в качестве результата:

```
Route::get('users/{user}/mail/preview', function (User $user) {
    return new App\Mail\SimpleMail($user->name);
});
```

Также можно получить содержание письма в формате HTML — например, для сохранения в файле. Для этого следует создать объект письма и вызвать у него `render()`. Пример:

```
$contents = (new App\Mail\Simplemail($user->name))->render();
```

23.5. События, генерируемые при отправке электронных писем

При отправке электронных писем генерируются два следующих события, классы которых объявлены в пространстве имен `Illuminate\Mail\Events`:

- `MessageSending` — генерируется перед отправкой письма. Вернув из обработчика значение `false`, можно отменить его отправку;
- `MessageSent` — генерируется после отправки письма.

Оба класса поддерживают свойства: `message` (хранит объект электронного письма) и `data` (ассоциативный массив со сведениями о письме).

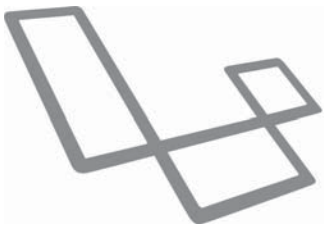
23.6. Доступ к письмам, отправленным посредством службы *array*

В *разд. 23.1* описывалась служба отправки писем `array`, помещающая все отсылаемые письма в массиве, который содержится в закрытом свойстве объекта, представляющего эту службу. Извлечь этот массив можно, написав следующее выражение:

```
$messages = Mail::mailer('array')->getSymfonyTransport()->messages());
```

Метод `getSymfonyTransport()` возвращает объект класса `Illuminate\Mail\Transport\ArrayTransport`, представляющий саму службу `array`. Метод `messages()`, вызванный у этой службы, возвращает массив писем.

Следует помнить, что служба `array` сохраняет отправленные письма только до поступления следующего клиентского запроса, после чего безвозвратно удаляет их.



ГЛАВА 24

Оповещения

Оповещение — это электронное сообщение, которое может быть отправлено по разным каналам: электронной почтой, в качестве SMS, через службу Slack или даже записано в таблицу базы данных. Оповещение реализуется в виде класса.

Обычно оповещение отсылается какому-либо объекту модели, представляющему запись из базы данных, — *адресату* (часто в качестве адресата выступает запись модели пользователя `User`). Фреймворк получает из этой записи указания по дальнейшей пересылке оповещения: адрес электронной почты, номер телефона или интернет-адрес клиента службы Slack, — после чего, руководствуясь этими сведениями, пересылает оповещение. Также возможна отправка оповещения по произвольному адресу, не записанному в базе данных.

24.1. Создание оповещений

Новый класс оповещения создается набором команды:

```
php artisan make:notification <имя класса оповещения> ↳
[--markdown=<путь к шаблону Markdown>] [--force]
```

Поддерживаются командные ключи:

- `--markdown` — создает класс, изначально предназначенный для генерирования почтовых оповещений на основе шаблонов, написанных на языке Markdown, и шаблон такого оповещения, сохраняемый по заданному *пути*, который отсчитывается от папки resources\views;
- `--force` — принудительно создает класс оповещения, даже если одноименный модуль уже существует.

Класс оповещения объявляется в пространстве имен `App\Notifications` (соответствующая папка создается автоматически) как производный от класса `Illuminate\Notifications\Notification`. Изначально он содержит трейт `Illuminate\Bus\Queueable`, позволяющий создавать отложенные оповещения (будут описаны в *разд. 25.5.3*). Если оповещение не предполагается делать отложенным, трейт может быть удален.

В классе оповещения исходно присутствуют четыре метода:

- конструктор — изначально «пустой». Может быть использован для получения каких-либо данных, нужных для работы, в том числе посредством внедрения зависимостей;
- `via()` — с единственным параметром получает объект-адресат и должен возвращать массив обозначений каналов, по которым будет пересылаться текущее оповещение. Поддерживаются следующие каналы: `'mail'` (электронная почта), `'vonage'` (SMS, в предыдущих версиях Laravel этот канал обозначался `'nexmo'`), `'slack'` (одноименная интернет-служба) и `'database'` (таблица в базе данных). Изначально возвращает массив с единственным элементом `'mail'`;
- `toMail()` — с единственным параметром получает объект-адресат и должен возвращать объект сгенерированного почтового оповещения. Изначально возвращает тестовое англоязычное письмо с двумя абзацами и гиперссылкой на «корень» сайта;
- `toArray()` — с единственным параметром принимает объект-адресат и должен возвращать ассоциативный массив со сведениями для записи в базу данных и пересылки по каналам вещания (которые будут описаны в *главе 31*). Изначально возвращает «пустой» массив.

В листинге 24.1 показан код оповещения `App\Notifications\SimpleNotification`, отправляющего пользователю по электронной почте сообщение с количеством объявлений в базе данных.

Листинг 24.1. Код оповещения `App\Notifications\SimpleNotification`

```
namespace App\Notifications;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;
use App\Models\Bb;
class SimpleNotification extends Notification {
    use Queueable;

    public function __construct() { }

    public function via($notifiable) {
        return ['mail'];
    }

    public function toMail($notifiable) {
        return (new MailMessage)
                    ->subject('Доска объявлений')
                    ->greeting('Уважаемый ' . $notifiable->name . '!')
                    ->line('В базе данных сейчас ' . Bb::count() .
                           ' объявлений.')
                    ->action('Посетите наш сайт', url('/'))
                    ->line('Спасибо за внимание.')
                    ->salutation('До свидания!');
    }
```

```
    public function toArray($notifiable) {
        return [];
    }
}
```

Это оповещение можно отправить текущему пользователю, вставив в действие контроллера следующий код:

```
use Illuminate\Support\Facades\Auth;
use App\Notifications\SimpleNotification;
. . .
if (Auth::check())
    Auth::user()->notify(new SimpleNotification);
```

24.2. Написание оповещений

24.2.1. Почтовые оповещения

Для отправки оповещений посредством электронной почты следует настроить подсистему электронной почты (см. *разд. 23.1*).

24.2.1.1. Генерирование простых почтовых оповещений

Сначала следует убедиться, что метод `via()` класса оповещения возвращает массив с обозначением `'mail'`.

Почтовое оповещение генерируется в методе `toMail()` класса оповещения. В качестве единственного параметра он принимает объект-адресат, а в качестве результата должен возвращать объект класса `Illuminate\Notifications\Messages\MailMessage`, представляющий содержимое почтового оповещения.

Объект содержимого оповещения поддерживает ряд представленных далее методов, предназначенных для создания различных фрагментов этого содержимого: приветствия, абзацев текста, гиперссылок и заключения. Эти методы возвращают текущий объект содержимого, вследствие чего их вызовы можно записывать цепочкой:

- `subject(<`*тема*`>)` — задает *тему* текущего почтового оповещения;
- `greeting(<`*приветствие*`>)` — задает *приветствие*;
- `line(<`*абзац*`>)` — добавляет в оповещение *абзац* обычного текста;
- `with()` — то же самое, что и `line()`;
- `lines(<`*массив с абзацами*`>)` — добавляет в оповещение абзацы, содержащиеся в заданном *массиве*;
- `action(<`*текст*`>, <`*интернет-адрес*`>)` — создает гиперссылку с заданными *текстом* и *интернет-адресом*. Гиперссылка в оповещении выводится в виде кнопки;
- `salutation(<`*заключение*`>)` — задает *заключение*;
- `level(<`*уровень*`>)` — задает *уровень* оповещения в виде строки `'info'` (информация нейтрального плана), `'success'` (уведомление об успехе) или `'error'` (сообщение об ошибке). По умолчанию используется уровень `'info'`.

Уровень оповещения задает цвет гиперссылки: серый (`'info'`), зеленый (`'success'`) или красный (`'error'`). Пример:

```
return (new MailMessage)->level('success')
                        ->line('У вас все получилось')
                        ->action('Посетите наш сайт', url('/'));
```

- `success()` — задает уровень оповещения `'success'`;
- `error()` — задает уровень оповещения `'error'`.

  По умолчанию оповещения отправляются посредством почтовой службы, указанной в настройках как используемая по умолчанию;
- `mailer(<имя службы>)` — указывает использовать для отправки текущего оповещения службу с заданным *именем*:

```
return (new MailMessage)->mailer('postmark')
                        ->subject('Доска объявлений')
                        . . .
```

Объект содержимого почтового оповещения также поддерживает методы: `from()`, `replyTo()`, `cc()`, `bcc()`, `attach()`, `attachData()` и `priority()`, описанные в *главе 23*:

```
return (new MailMessage)->from('support@bboard.ru', 'Служба поддержки')
                        ->attach('c:/site/mail/attaches/license.doc')
                        ->line('Получите лицензионное соглашение');
```

Простое почтовое оповещение подобного рода имеет строго заданную разметку. Изменить ее можно, исправив шаблон простого оповещения. Изначально этот шаблон находится в составе фреймворка, и перед правкой его необходимо извлечь, набрав команду:

```
php artisan vendor:publish --tag=laravel-notifications
```

После ее выполнения будет создана папка resources\views\vendor\notifications с шаблоном простого оповещения email.blade.php. Он написан на языке Markdown и использует для вывода содержимого письма следующие переменные, создаваемые в контексте этого шаблона:

- `subject` — тема письма;
- `greeting` — приветствие;
- `introLines` — абзацы, созданные перед гиперссылкой;
- `actionText` — текст гиперссылки;
- `actionUrl` — интернет-адрес гиперссылки;
- `outroLines` — абзацы, созданные после гиперссылки;
- `salutation` — заключение;
- `level` — уровень оповещения;
- `displayableActionUrl` — интернет-адрес гиперссылки с удаленными префиксами `mailto:` и `tel:`. Предназначен для вывода на экран.

24.2.1.2. Генерирование почтовых оповещений на основе текстовых и HTML-шаблонов

Сгенерировать почтовое оповещение также можно на основе произвольного шаблона, вызвав у объекта содержимого почтового оповещения метод `view()`:

```
view(<сведения о шаблонах>[, <контекст шаблона>=[]])
```

В качестве *сведений о шаблонах* можно указать:

- путь к шаблону в формате HTML — тогда будет сгенерировано оповещение в формате HTML:

```
return (new MailMessage)->view('mails.notice-html', ['user' => $user]);
```

- массив из путей к шаблонам в форматах HTML и текстовом — тогда будет сгенерировано составное оповещение, содержащее текстовую и HTML-части:

```
return (new MailMessage)->view(['mails.notice-html',
                                'mails.notice-text'],
                               ['user' => $user]);
```

Также можно вернуть из метода `toMail()` объект класса письма (см. *главу 23*). Для указания адреса получателя следует использовать метод `to()` класса письма, имеющего тот же формат вызова, что и одноименный метод фасада `Mail` (см. *разд. 23.3*). Пример:

```
use App\Mail\SimpleMail;
class SimpleNotification extends Notification {
    . . .
    public function toMail($notifiable) {
        return (new SimpleMail($notifiable->name))
                  ->to($notifiable->email);
    }
    . . .
}
```

24.2.1.3. Генерирование почтовых оповещений на основе Markdown-шаблонов

Чтобы сгенерировать почтовое оповещение на основе произвольного шаблона, написанного на языке Markdown, следует вызвать у объекта содержимого почтового оповещения метод `markdown()`:

```
markdown(<путь к шаблону>[, <контекст шаблона>=[]])
```

Пример:

```
return (new MailMessage)->markdown('mails.notice-markdown',
                                   ['user' => $user]);
```

В Markdown-шаблонах можно использовать компоненты, описанные в *разд. 23.2.4.2*.

Изменить разметку и оформление оповещения, генерируемого на основе Markdown-шаблона, можно, воспользовавшись приемом, рассмотренным в *разд. 23.2.4.3*.

Если было создано несколько таблиц стилей, описывающих различные варианты оформления оповещений, можно указать таблицу стилей для текущего оповещения не-

посредственно при его создании. Для этого достаточно вызвать у объекта содержимого оповещения метод `theme(<имя таблицы стилей>)`, где *имя таблицы стилей* указывается без расширения. Пример:

```
return (new MailMessage)
            ->theme('notice')
            ->markdown('mails.notice-markdown', ['user' => $user]);
```

24.2.1.4. Указание адреса получателя

По умолчанию Laravel извлекает адрес электронной почты получателя почтового оповещения из свойства `email` объекта-адресата.

Можно указать фреймворку другой адрес для посылки оповещения, объявив в классе объекта-адресата общедоступный метод `routeNotificationForMail()`. В качестве параметра он должен принимать объект оповещения и возвращать один из двух результатов:

- строку с адресом электронной почты;
- ассоциативный массив из одного элемента, ключ которого задаст адрес, а значение — имя получателя:

```
class User extends Authenticatable {
    . . .
    public function routeNotificationForMail($notification) {
        return [$this->email_address => $this->username];
    }
}
```

Если в качестве оповещения используется электронное письмо (см. *разд. 24.2.1.2*), адрес получателя указывается при создании этого письма вызовом метода `to()`.

24.2.2. SMS-оповещения

Отправка SMS-оповещений выполняется через интернет-службу Vonage (**https://www.vonage.ru/**).

24.2.2.1. Подготовительные действия и настройка службы SMS-оповещений

Для успешной отправки SMS-оповещений следует установить дополнительные библиотеки, набрав команду:

```
composer require laravel/vonage-notification-channel guzzlehttp/guzzle
```

Далее необходимо указать в файле локальных настроек .env следующие настройки:

- `VONAGE_KEY` — идентификатор приложения Nexmo;
- `VONAGE_SECRET` — секретный ключ приложения Nexmo;
- `VONAGE_SMS_FROM` — телефонный номер отправителя SMS.

24.2.2.2. Генерирование произвольных SMS-оповещений

Сначала нужно исправить метод `via()` класса оповещения таким образом, чтобы он возвращал массив с обозначением `'vonage'`:

```
class FailureNotification extends Notification {
    . . .
    public function via($notifiable) {
        return ['vonage'];
    }
    . . .
}
```

SMS-оповещение генерируется в методе `toVonage()` класса оповещения (в предыдущих версиях фреймворка этот метод назывался `toNexmo()`). Единственным параметром он принимает объект-адресат и должен возвращать объект класса `Illuminate\Notifications\Messages\VonageMessage`, представляющий содержимое SMS-оповещения.

Объект содержимого поддерживает четыре метода, используемых для создания SMS-оповещения:

- `content(<`*содержимое*`>)` — задает *содержимое* оповещения:

```
use Illuminate\Notifications\Messages\VonageMessage;
class FailureNotification extends Notification {
    . . .
    public function toVonage($notifiable) {
        return (new VonageMessage)
             ->content('There is failure on site!');
    }
    . . .
}
```

- `unicode()` — указывает фреймворку, что текущее оповещение написано в кодировке Unicode:

```
public function toVonage($notifiable) {
    return (new VonageMessage)
         ->content('Авария на сайте!')->unicode();
}
```

- `from(<`*номер телефона*`>)` — задает у текущего оповещения *номер телефона* отправителя, который следует указывать в виде строки. Заданный номер перекрывает заданный в локальной настройке `VONAGE_SMS_FROM`;
- `clientReference(<`*идентификатор*`>)` — задает *идентификатор* получателя сообщения, представляемый в виде строки длиной не более 40 символов. Такие идентификаторы используются службой Vonage при формировании статистики рассылки SMS, сгруппированной по получателям. Пример:

```
public function toVonage($notifiable) {
    return (new VonageMessage)
         ->content(...)->clientReference((string) $notifiable->id);
}
```

24.2.2.3. Указание телефона получателя

Чтобы фреймворк смог получить телефон получателя SMS-оповещения, в классе объекта-адресата следует объявить общедоступный метод `routeNotificationForVonage()`. В качестве параметра он должен принимать объект оповещения и возвращать телефонный номер получателя в виде строки. Пример:

```
class User extends Authenticatable {
    . . .
    public function routeNotificationForVonage($notification) {
        return $this->phone;
    }
}
```

24.2.3. Slack-оповещения

Slack (**https://slack.com/**) — популярная интернет-служба для организации совместной работы.

Для успешной отправки оповещений в службу Slack следует установить дополнительную библиотеку, набрав команду:

```
composer require laravel/slack-notification-channel
```

Никаких настроек для использования службы Slack задавать не нужно.

24.2.3.1. Генерирование Slack-оповещений

Необходимо исправить метод `via()` класса оповещения таким образом, чтобы он возвращал обозначение `'slack'`.

Slack-оповещение генерируется в методе `toSlack()` класса оповещения. В качестве единственного параметра он принимает объект-адресат, а качестве результата должен возвращать объект класса `Illuminate\Notifications\Messages\SlackMessage`, представляющий содержимое Slack-оповещения.

Объект содержимого поддерживает следующие методы:

- `content(<`*содержимое*`>)` — задает *содержимое* текущего оповещения:

```
use Illuminate\Notifications\Messages\SlackMessage;
class FailureNotification extends Notification {
    . . .
    public function via($notifiable) {
        return ['slack'];
    }

    public function toSlack($notifiable) {
        return (new SlackMessage)->content('Авария на сайте!');
    }
    . . .
}
```

- `from(<`*имя отправителя*`>[, <`*обозначение значка*`>=null])` — задает *имя отправителя* и необязательное *обозначение значка* текущего оповещения;

- `to(<имя получателя>)` — задает *имя получателя* — пользователя или канала — текущего сообщения. Пример:

```
return (new SlackMessage)->from('Ghost', ':ghost:')->to('#emergency')
                         ->content('Авария на сайте!');
```

- `image(<путь к файлу>)` — задает *путь к файлу* со значком для текущего сообщения. Заданный вызовом этого метода значок будет использован вместо значка, указанного в вызове метода `from()`. Пример:

```
return (new SlackMessage)->from('Ghost')
                         ->image('c:/site/icons/horrible-ghost.png')
                         ->content('Авария на сайте!');
```

- `success()` — задает для текущего оповещения уровень `'success'` (сообщает, что какое-либо действие успешно выполнено);
- `warning()` — задает для текущего оповещения уровень `'warning'` (в процессе выполнения действия возникли проблемы, не требующие, однако, вмешательства администрации сайта или пользователя);
- `error()` — задает для текущего оповещения уровень `'error'` (фатальная ошибка);
- `info()` — задает для текущего оповещения уровень `'info'` (информация нейтрального плана).

24.2.3.2. Добавление вложений

Обычное Slack-оповещение — это просто текст с необязательным значком. Добавить в оповещение какие-либо дополнительные данные можно, создав вложение.

Вложения создаются вызовом у объекта содержимого оповещения метода `attachment(<анонимная функция>)`. *Анонимная функция* должна принимать единственный параметр — «пустой» объект вложения и создавать вложение, вызывая у этого объекта различные методы.

Вот методы, поддерживаемые классом вложения:

- `title(<заголовок>[, <интернет-адрес>=null])` — создает *заголовок* текущего вложения, выводимый в его начале. Если дополнительно указать *интернет-адрес*, заголовок станет гиперссылкой, указывающей на этот интернет-адрес;
- `content(<содержимое>)` — создает *содержимое* текущего вложения. Оно будет выведено непосредственно под заголовком. Пример:

```
return (new SlackMessage)->content('Авария на сайте!')
            ->attachment(function ($attach) {
                $attach->title('Сбой в веб-сервере')
                       ->content('Не та конфигурация');
            });
```

Заголовок и содержимое должны присутствовать в любом вложении. Напротив, элементы, создаваемые приведенными далее методами, являются необязательными:

- `pretext(<вводной текст>)` — создает *вводной текст*, располагаемый над вложением;

- `author()` — задает *имя автора* вложения, располагаемое в самом начале вложения, перед заголовком:

```
author(<имя автора>[, <интернет-адрес сайта>=null[,
                      <интернет-адрес значка>=null]])
```

  Если дополнительно указан *интернет-адрес сайта* автора, выводимое над вложением имя автора превратится в гиперссылку, указывающую на сайт автора. Если также указан *интернет-адрес значка* автора, этот значок будет выводиться слева от имени автора. Пример:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->author('Head Developer',
                           'http://bboard.ru/head/',
                           'http://bboard.ru/icons/head.png');
});
```

- `field(<заголовок поля>[, <содержимое поля>=''])` — добавляет в текущее вложение поле с заданными *заголовком* и *содержимым*:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->field('Код ошибки', '404')
                  ->field('Запрошенный путь', request()->path());
});
```

  Можно добавить произвольное количество полей. Поля выводятся под содержимым вложения в две колонки;

- `fields(<ассоциативный массив с полями>)` — добавляет в текущее вложение поля из заданного *массива*. Ключи элементов *массива* станут заголовками полей, а значения элементов — содержимым этих полей. Пример:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->fields(['Код ошибки' => '404',
                            'Запрошенный путь'=>request()->path()]);
});
```

- `footer(<поддон>)` — создает *поддон*, выводимый под содержимым вложения;
- `footerIcon(<интернет-адрес значка>)` — выводит левее поддона значок с заданным *интернет-адресом*;
- `fallback(<заключение>)` — создает *заключение*, располагаемое под вложением;
- `image(<интернет-адрес>)` — задает *интернет-адрес* изображения, которое будет выведено под поддоном вложения крупным планом;
- `thumb(<интернет-адрес>)` — задает *интернет-адрес* изображения, которое будет выведено правее заголовка вложения в виде миниатюры.

  Если у вложения одновременно вызвать методы `image()` и `thumb()`, будет создано только изображение, выводимое крупным планом (как будто был вызван один метод `image()`);

- `action()` — создает гиперссылку с заданными *текстом* и *интернет-адресом*, имеющую вид кнопки и выводимую внизу вложения:

```
action(<текст>, <интернет-адрес>[, <стиль>=''])
```

Пример:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->action('Laravel', 'https://laravel.com/');
});
```

- `timestamp(<временна́я отметка>)` — задает *временну́ю отметку*, выводимую правее поддона. *Временна́я отметка* может быть задана в любом формате, поддерживаемом PHP. Пример:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->timestamp(now());
});
```

- `color(<цвет>)` — задает *цвет*, которым будет выводиться вертикальная линия слева от вложения, в виде строки: `'good'` (зеленый цвет), `'danger'` (красный), `'warning'` (желтый) или «пустая» строка (серый);
- `markdown(<массив обозначений>)` — указывает, в каких частях вложения будет обрабатываться форматирование на языке Markdown. В задаваемом массиве можно указать элементы: `'pretext'` (форматирование будет обрабатываться во вводном тексте), `'fields'` (в полях) и `'text'` (в остальных частях). Пример:

```
return (new SlackMessage)->attachment(function ($attach) {
    $attach-> ... ->markdown(['text', 'fields']);
});
```

24.2.3.3. Указание интернет-адреса получателя

Чтобы фреймворк смог получить интернет-адрес получателя Slack-оповещения, в классе объекта-адресата следует объявить общедоступный метод `routeNotificationForSlack()`. В качестве параметра он должен принимать объект оповещения и возвращать интернет-адрес клиента службы Slack в виде строки. Пример:

```
class User extends Authenticatable {
    . . .
    public function routeNotificationForSlack($notification) {
        return 'https://hooks.slack.com/services/...';
    }
}
```

24.2.4. Табличные оповещения

Табличные оповещения записываются в специально созданную таблицу базы данных и могут быть впоследствии прочитаны.

24.2.4.1. Создание таблицы для хранения табличных оповещений

Для формирования миграции, создающей таблицу оповещений, следует набрать команду:

```
php artisan notifications:table
```

В результате в папке database\migrations появится модуль миграции *<временна́я отметка>*_create_notifications_table.php. После этого необходимо выполнить миграции.

Таблица `notifications`, создаваемая в результате выполнения миграции, содержит следующие поля:

- `id` — строковое, длиной 255 символов — ключевое, хранящее универсальный уникальный идентификатор;
- `type` — строковое, длиной 255 символов — тип сохраненного оповещения в виде пути к его классу;
- `notifiable` — полиморфное поле внешнего ключа — полиморфная связь с объектом-адресатом;
- `data` — текстовое — содержимое сохраненного оповещения в формате JSON;
- `read_at` — временна́я отметка, не обязательная к заполнению, — *отметка чтения*, указывающая, когда это оповещение было прочитано;
- поля отметок создания и правки.

24.2.4.2. Генерирование табличных оповещений

Необходимо исправить метод `via()` класса оповещения таким образом, чтобы он возвращал обозначение `'database'`.

Табличные оповещения можно генерировать в одном из двух методов класса оповещения:

- `toDatabase()` — генерирует только табличное оповещение;
- `toArray()` — генерирует оповещение, отправляемое по каналам вещания (о них будет рассказано в *главе 31*). Если метод `toDatabase()` отсутствует, также генерирует табличное оповещение.

Если оповещения, записываемые в таблицу и отправляемые через каналы вещания, должны быть одного формата, или если вещание в проекте не реализуется, можно объявить лишь метод `toArray()` (он, кстати, уже присутствует во вновь созданном классе оповещения). В противном случае следует объявить оба метода.

Оба метода должны принимать в качестве параметра объект-адресат и возвращать массив с данными, сохраняемыми после преобразования в формат JSON в поле `data` таблицы `notifications`.

Пример:

```
class LoginNotification extends Notification {
    . . .
    public function via($notifiable) {
        return ['database'];
    }
    . . .
    public function toArray($notifiable) {
        return ['name' => $notifiable->name,
                'email' => $notifiable->email];
    }
}
```

24.2.5. Оповещения, отправляемые по нескольким каналам

Чтобы создать оповещение, отправляемое сразу по нескольким каналам (например, по почте и в качестве SMS), необходимо:

- в методе `via()` класса оповещения — вернуть массив с обозначениями всех необходимых каналов:

```
public function via($notifiable) {
    return ['mail', 'vonage'];
}
```

  Поскольку этот метод получает с параметром объект-адресат, можно возвращать массив с разными обозначениями каналов, в зависимости от значений свойств объекта-адресата:

```
public function via($notifiable) {
    if ($notifiable->sendMeSMS) {
        return ['mail', 'vonage'];
    } else {
        return ['mail'];
    }
}
```

- в классе оповещения — объявить все необходимые методы, генерирующие оповещения (`toMail()`, `toVonage()` и др.).

24.3. Отправка оповещений

Отправить оповещение можно двумя способами:

- вызвав метод `send()` у фасада `Illuminate\Support\Facades\Notification`:

```
send(<адресат>|<массив адресатов>, <объект оповещения>)
```

  Можно указать как один объект-*адресат*:

```
use Illuminate\Support\Facades\Notification;
. . .
Notification::send(Auth::user(), new FailureNotification);
```

  так и *массив* или коллекцию объектов-адресатов:

```
Notification::send(User::where('admin', true)->get(),
                   new FailureNotification);
```

- если класс объекта-адресата содержит трейт `Illuminate\Notifications\Notifiable` (а модель пользователя `User` содержит его изначально) — вызвав у объекта-адресата метод `notify(<объект оповещения>)`:

```
Auth::user()->notify(new SimpleNotification);
```

24.3.1. Отправка оповещений произвольным получателям

Если получатель оповещения не записан в списке пользователей или вообще где-либо в базе данных, для отправки ему оповещения следует применить метод `route()` фасада `Notification`:

```
route(<обозначение канала отправки>, <адрес отправки>)
```

В качестве *адреса отправки*, в зависимости от заданного *канала*, следует задать адрес электронной почты, номер телефона или интернет-адрес пользователя службы Slack.

Фасад `Notification` при обращении выдает объект службы отправки оповещений произвольным получателям. Тот же объект возвращает и метод `route()`. Таким образом, можно отправить одно оповещение сразу по нескольким каналам, записывая вызовы метода `route()` цепочкой.

Чтобы выполнить отправку оповещения после указания сведений о получателе, достаточно вызвать у объекта службы отправки, возвращенного последним в цепочке вызовом метода `route()`, метод `notify()` (был описан в *разд. 24.3*). Пример:

```
Notification::route('mail', 'freelancer@mail.ru')
            ->route('vonage', '0987654321')
            ->route('slack', 'https://hooks.slack.com/services/...')
            ->notify(new SomeNotification);
```

24.4. Предварительный просмотр почтовых оповещений

Чтобы вывести почтовое оповещение в веб-обозревателе, достаточно в действии контроллера создать объект оповещения, вызвать у него метод `toMail()` и вернуть возвращенный им результат:

```
Route::get('users/{user}/notification/preview', function (User $user) {
    return (new App\Notifications\SimpleNotification)->toMail($user);
});
```

24.5. Работа с табличными оповещениями

Laravel позволяет извлечь все табличные оповещения, отправленные заданному объекту-адресату, просмотреть их и пометить как прочитанные.

Чтобы, имея объект-адресат, получить все отправленные ему табличные оповещения, класс объекта-адресата должен содержать трейт `Illuminate\Notifications\Notifiable` (модель пользователя `User` содержит его изначально). Этот трейт добавляет в класс метод `notifications()`, устанавливающий «прямую» полиморфную связь с таблицей оповещений. Следовательно, для извлечения оповещений можно использовать способы, описанные в *разд. 16.4.3*. Пример:

```
>>> use App\Models\User;
>>> // Какие оповещения отправлены пользователю admin?
>>> $user = User::firstWhere('name', 'admin');
>>> $notifications = $user->notifications;
>>> foreach ($notifications as $n) {
...     echo $n->type, "\r\nname: ", $n->data['name'], ', email: ',
...          $n->data['email'], "\r\n";
... }
App\Notifications\SimpleNotification
name: admin, email: admin@bboard.ru
>>> // Всего одно оповещение
```

Как видно из приведенного примера, данные, сохраненные в поле `data` в формате JSON, при извлечении преобразуются в обычный ассоциативный массив PHP, что позволяет с легкостью обработать их.

Также трейт `Notifiable` добавляет классу поддержку следующих методов:

- `unreadNotifications()` — устанавливает «прямую» полиморфную только с непрочитанными оповещениями (в которых поле `read_at` хранит `null`):

```
// Перебираем только непрочитанные оповещения
foreach ($user->unreadNotification as $n) {
    . . .
}
```

- `readNotifications()` — устанавливает «прямую» полиморфную только с прочитанными оповещениями (в которых поле `read_at` хранит значение, отличное от `null`).

Оповещение, извлекаемое из таблицы, представляется моделью `Illuminate\Notifications\DatabaseNotification`. С ее помощью можно перебрать перечень всех оповещений:

```
use Illuminate\Notifications\DatabaseNotification;
$notifications = DatabaseNotification::all();
foreach ($notifications as $n) {
    $username = $n->data['name'];
    $email = $n->data['email'];
    . . .
}
```

Помимо свойств и методов, поддерживаемых всеми моделями Laravel, модель оповещений дополнительно содержит пять полезных методов:

- `unread()` — возвращает `true`, если текущее оповещение еще не прочитано, и `false` — в противном случае;
- `read()` — возвращает `true`, если текущее оповещение уже прочитано, и `false` — в противном случае;
- `markAsRead()` — помечает текущее оповещение как прочитанное;
- `markAsUnread()` — помечает текущее оповещение как непрочитанное;
- `notifiable()` — создает «обратную» полиморфную связь с объектом-адресатом:

```
foreach ($notifications as $n) {
    // Получаем пользователя-адресата
    $notifiable = $n->notifiable;
```

```
        // Получаем имя пользователя-адресата
        $username = $notifiable->name;
        . . .
    }
```

Коллекция объектов оповещений также поддерживает методы `markAsRead()` и `markAsUnread()`, помечающие соответственно как прочитанные и непрочитанные все оповещения, что есть в коллекции:

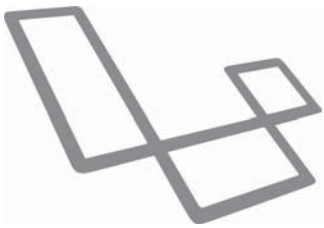
```
$user->notifications()->markAsRead();
```

24.6. События, генерируемые при отправке оповещений

При отправке оповещений генерируются три следующих события, классы которых объявлены в пространстве имен `Illuminate\Notifications\Events`:

- `NotificationSending` — генерируется перед отправкой оповещения. Поддерживаются свойства:
  - `notifiable` — объект-адресат;
  - `notification` — объект оповещения;
  - `channel` — строковое обозначение канала отправки.

  Вернув из обработчика этого события значение `false`, можно отменить отправку оповещения;
- `NotificationSent` — генерируется после отправки оповещения. Поддерживаются свойства, содержащиеся в классе `NotificationSending`, и дополнительно свойство `response`, которое хранит ответ, переданный службой отправки оповещений;
- `NotificationFailed` — генерируется при возникновении ошибки при отправке оповещения. Поддерживаются свойства, содержащиеся в классе `NotificationSending`, и дополнительно свойство `data`, которое хранит ассоциативный массив со сведениями о возникшей ошибке.



ГЛАВА 25

Очереди и отложенные задания

В процессе обработки клиентских запросов часто приходится выполнять достаточно длительные операции: отправку электронного письма или оповещения, обработку большого количества записей в базе данных, удаление множества файлов и др. Такие операции способны существенно замедлить генерирование результирующей веб-страницы и снизить отзывчивость сайта.

Однако Laravel позволяет выполнить такую долгую операцию в другом процессе отдельным обработчиком, не мешая обработке запросов. Операции, предназначенные для выполнения в другом процессе, временно сохраняются в особом хранилище — *очереди* и носят название *отложенных заданий*.

Если отложенное задание выполнить не удалось вследствие его «зависания» или сгенерированного в его коде исключения, задание считается *проваленным* и помещается в отдельный список. Впоследствии обработчик будет предпринимать попытки все-таки выполнить проваленные задания, и, возможно, какая-то из них увенчается успехом.

25.1. Настройка подсистемы очередей

Laravel может хранить очереди в разных службах: нереляционной базе данных Redis, реляционной базе данных одного из поддерживаемых форматов, сторонних интернет-службах. В каждой службе можно создать произвольное количество независимых очередей для хранения заданий разного рода (так, в одной очереди можно хранить задания на удаление файлов, а в другой — задания на рассылку писем).

25.1.1. Настройка самих очередей

Настройки собственно очередей хранятся в модуле config\queue.php:

- `connections` — ассоциативный массив служб, в которых будут храниться очереди. Ключи элементов этого массива задают имена служб, а значения элементов представляют собой вложенные ассоциативные массивы, содержащие настройки этих служб. Поддерживаются следующие настройки:
  - `driver` — тип службы. Доступны типы:
    - `sync` — очередь вообще не формируется, а отложенные задания выполняются немедленно после их создания. Используется только при отладке;

- `database` — таблица в обычной реляционной базе данных;
- `beanstalkd` — «легкий» сервер очередей Beanstalkd (**https://beanstalkd.github.io/**). Требует установки дополнительной библиотеки подачей команды:

  ```
  composer require pda/pheanstalk ~4.0
  ```

- `sqs` — служба Amazon SQS. Требует установки дополнительной библиотеки, для чего нужно подать команду:

  ```
  composer require aws/aws-sdk-php ~3.0
  ```

- `redis` — нереляционная база данных Redis. Для ее использования следует установить дополнительную библиотеку predis, набрав команду:

  ```
  composer require predis/predis ~1.0
  ```

  Также можно установить расширение PHP под названием phpredis (**https://github.com/phpredis/phpredis**). Оно позволит достичь более высокой производительности, но требует конфигурирования в файле настроек PHP, что может оказаться неприемлемым при публикации сайта на стороннем хостинге;
- `null` — очередь не создается, и отложенные задания не выполняются. Используется только при отладке.

Следующие настройки используются всеми службами, кроме `sync` и `null`:

- `queue` — имя очереди по умолчанию. В нее будут помещаться задания, если при их создании очередь не была указана явно. Значения по умолчанию этой настройки различаются у разных служб и будут описаны далее;
- `retry_after` — максимальное время, отводимое на выполнение отложенного задания, в секундах (по умолчанию: 90);
- `after_commit` — учитывается при выполнении операций с базами данных. Если `true`, очередное задание будет выполнено строго после подтверждения текущей транзакции. Если `false`, очередное задание может начать выполняться в произвольный момент времени, в том числе и до завершения транзакции. По умолчанию: `false`.

Следующие настройки используются службой `database`:

- `connection` — имя базы данных, присутствующей в массиве из рабочей настройки `database.connections` (см. *разд. 3.4.2.4*). Если настройка не указана, используется база данных по умолчанию;
- `table` — имя таблицы, в которой будет храниться очередь (по умолчанию: `jobs`);
- `queue` — значение по умолчанию: `default`.

Следующие настройки используются службой `beanstalkd`:

- `host` — интернет-адрес компьютера с установленным сервером очередей Beanstalkd (по умолчанию: **localhost**);
- `queue` — значение по умолчанию: `default`;
- `block_for` — промежуток времени между последовательными опросами сервера очередей в секундах (по умолчанию: 0).

Следующие настройки используются службой `sqs`:

- `key` — ключ доступа. Значение берется из локальной настройки `AWS_ACCESS_KEY_ID`, присутствующей в файле .env, но изначально «пустой»;
- `secret` — секретный ключ. Значение берется из локальной настройки `AWS_SECRET_ACCESS_KEY`, присутствующей в файле .env, но изначально «пустой»;
- `prefix` — базовый интернет-адрес службы, включающий обозначение региона и регистрационный идентификатор. Значение берется из локальной настройки `SQS_PREFIX`, изначально отсутствующей. По умолчанию: **https://sqs.us-east-1.amazonaws.com/your-account-id**;
- `queue` — значение берется из локальной настройки `SQS_QUEUE`, изначально отсутствующей в файле .env. Значение по умолчанию: `default`;
- `suffix` — суффикс, добавляемый к имени очереди. Значение берется из локальной настройки `SQS_SUFFIX`, изначально отсутствующей в файле .env;
- `region` — обозначение региона. Значение берется из локальной настройки `AWS_DEFAULT_REGION`. По умолчанию: `us-east-1`.

Следующие настройки используются службой `redis`:

- `connection` — обозначение соединения с Redis (настройка соединений с этой СУБД будет описана далее). Значение по умолчанию: `default`;
- `queue` — значение берется из локальной настройки `REDIS_QUEUE`, изначально отсутствующей в файле .env. Значение по умолчанию: `default`;
- `block_for` — промежуток времени между последовательными опросами базы данных на предмет появления в очереди нового задания. Указывается в виде целого числа в секундах. Если указать `0` или `null`, Redis будет сама посылать фреймворку сообщения о появлении в очереди новых заданий. По умолчанию: `null`.

Изначально присутствуют службы: `sync`, `database`, `beanstalkd`, `sqs` и `redis`;

- `default` — имя службы очередей, используемой по умолчанию, если при создании задания служба не была указана явно. Значение берется из локальной настройки `QUEUE_CONNECTION`, имеющей значение `sync`. По умолчанию: `sync`;
- `failed` — настройки таблицы, хранящей список проваленных заданий. Указываются в виде ассоциативного массива со следующими параметрами:
  - `driver` — тип службы, хранящей эту таблицу. Поддерживаются значения:
    - `database-uuids` — таблица в обычной реляционной базе данных;
    - `dynamodb` — нереляционная база данных Amazon DynamoDB;
    - `null` — проваленные задания нигде не хранятся. Использовать следует лишь в том случае, если проваленных заданий заведомо не будет.

    Значение берется из локальной настройки `QUEUE_FAILED_DRIVER`, изначально отсутствующей в файле .env. По умолчанию: `database-uuids`.

Следующая настройка используется всеми службами, кроме `null`:

- `table` — имя самой таблицы, хранящей список проваленных заданий (по умолчанию: `failed_jobs`).

Следующая настройка используется службой `database-uuids`:

- `database` — имя базы данных, где хранится таблица со списком проваленных заданий. Должна быть указана какая-либо база данных из присутствующих в массиве из рабочей настройки `database.connections`. Значение берется из локальной настройки `DB_CONNECTION`. По умолчанию: `mysql`.

Следующие настройки используются службой `dynamodb`:

- `key` — ключ доступа. Значение берется из локальной настройки `AWS_ACCESS_KEY_ID`, присутствующей в файле .env, но изначально «пустой»;
- `secret` — секретный ключ. Значение берется из локальной настройки `AWS_SECRET_ACCESS_KEY`, присутствующей в файле .env, но изначально «пустой»;
- `region` — обозначение региона. Значение берется из локальной настройки `AWS_DEFAULT_REGION`. По умолчанию: `us-east-1`.

25.1.2. Подготовка таблиц для хранения отложенных заданий

Если для хранения очереди и (или) списка проваленных заданий используется обычная реляционная база данных, необходимо создать соответствующие таблицы.

Для создания миграции, формирующей таблицу очереди заданий, используется команда:

```
php artisan queue:table
```

Создаваемая миграция получает имя формата *<временна́я отметка>*_create_jobs_table.php.

Имя создаваемой таблицы считывается из рабочей настройки `queue.connections.database.table`. К сожалению, таблица всегда создается в базе данных по умолчанию, поэтому, если планируется поместить очередь в другой базе данных, придется исправить код созданной миграции вручную.

Миграция, создающая таблицу списка проваленных заданий, уже присутствует в любом вновь созданном проекте и имеет имя формата *<временна́я отметка>*_create_failed_jobs_table.php. Изначально она создает таблицу `failed_jobs` в базе данных по умолчанию.

Если миграция для создания таблицы проваленных заданий по какой-то причине была удалена или необходимо хранить проваленные задания в таблице с другим именем, новую миграцию можно создать набором команды:

```
php artisan queue:failed-table
```

Если список проваленных заданий планируется хранить в другой базе данных, потребуется внести соответствующие правки в код этой миграции.

Далее необходимо выполнить миграции, чтобы все нужные таблицы были созданы.

25.1.3. Настройка баз данных Redis

Если для хранения очереди применяется СУБД Redis, следует записать в конфигурации фреймворка все используемые базы данных этого формата. Необходимые настройки указываются в модуле config\database.php — в настройке `redis`, значением которой является ассоциативный массив. В этом массиве задаются как основные настройки, касающиеся самой СУБД, так и параметры отдельных баз данных.

Вот основные настройки Redis:

- `client` — библиотека, используемая для взаимодействия с Redis. Можно указать значения `phpredis` (одноименное расширение PHP) или `predis` (одноименная дополнительная библиотека, написанная на PHP). Значение берется из локальной настройки `REDIS_CLIENT`, изначально отсутствующей в файле .env. По умолчанию: `phpredis`;
- `options` — дополнительные параметры, указываемые в виде ассоциативного массива со следующими настройками:
  - `cluster` — имя кластера СУБД Redis, если таковой используется. Значение берется из локальной настройки `REDIS_CLUSTER`, изначально отсутствующей в файле .env. По умолчанию: `redis`;
  - `prefix` — префикс, добавляемый к именам записываемых значений, чтобы избежать их совпадений с именами значений, сохраняемых другими программами. Значение берется из локальной настройки `REDIS_PREFIX`, изначально отсутствующей в файле .env. По умолчанию: строка, составленная из названия проекта (берется из локальной настройки `APP_NAME`) и слова `database`, записанная в стиле snake_case.

Настройки отдельных баз данных также записываются в настройке `redis` как элементы ассоциативного массива. Их ключи задают имена баз данных, а значения представляют собой вложенные ассоциативные массивы с их настройками:

- `url` — интернет-адрес базы данных, включающий адрес сервера, имя базы данных, имя пользователя и пароль. Значение берется из локальной настройки `REDIS_URL`, изначально отсутствующей в файле .env;
- `host` — интернет-адрес хоста, на котором работает Redis. Значение берется из локальной настройки `REDIS_HOST`. По умолчанию: **127.0.0.1**;
- `port` — номер TCP-порта, через который работает Redis. Значение берется из локальной настройки `REDIS_PORT`. По умолчанию: 6379;
- `username` — регистрационное имя для подключения к Redis. Значение берется из локальной настройки `REDIS_USERNAME`;
- `password` — пароль для подключения к Redis. Значение берется из локальной настройки `REDIS_PASSWORD`. По умолчанию: `null` (без пароля);
- `database` — имя базы данных.

> ***Можно указать либо настройку `url`...***
>
> ...либо настройки: `host`, `port`, `password` и `database` — но не то и другое вместе.

Изначально созданы две базы данных:

- `default` — чье имя берется из локальной настройки `REDIS_DB`, изначально отсутствующей (по умолчанию: `0`);
- `cache` — чье имя берется из локальной настройки `REDIS_CACHE_DB`, изначально отсутствующей (по умолчанию: `1`).

Разумеется, можно добавить дополнительные базы данных.

25.2. Отложенные задания-классы

Наиболее часто отложенные задания оформляются в виде классов (*задания-классы*). Такие задания позволяют использовать развитые программные инструменты для управления выполнением заданий, обработки возникающих в них ошибок и др.

25.2.1. Создание отложенных заданий-классов

25.2.1.1. Создание отложенных заданий-классов: базовые инструменты

Для создания нового отложенного задания-класса нужно набрать команду:

```
php artisan make:job <имя класса задания> [--sync]
```

Командный ключ `--sync` будет рассмотрен позже.

Класс отложенного задания объявляется в пространстве имен `App\Jobs` (соответствующая папка создается автоматически) и не является ничьим подклассом. Он реализует интерфейс `Illuminate\Contracts\Queue\ShouldQueue`, помечающий класс как собственно отложенное задание, помещаемое в очередь. Помимо того, он включает следующие трейты:

- `Illuminate\Foundation\Bus\Dispatchable` — добавляет поддержку статических методов, запускающих выполнение задания;
- `Illuminate\Queue\InteractsWithQueue` — позволяет заданию взаимодействовать с очередью. Если таковое не предусматривается, может быть удален для экономии системных ресурсов;
- `Illuminate\Bus\Queueable` — реализует размещение задания в очереди. Ключевой трейт, который ни в коем случае нельзя удалять.

  При отдаче команды на выполнение задания создается его объект, который сериализуется в строковое представление и в таком виде записывается в очередь. Когда обработчик отложенных заданий (будет описан в *разд. 25.6.1*) извлекает задание из очереди, производится десериализация объекта в его изначальное состояние.

  Сериализации подлежат значения всех общедоступных и защищенных свойств объекта задания. Это позволяет сохранять в них нужные для последующего выполнения задания данные;
- `Illuminate\Queue\SerializesModels` — реализует сериализацию объектов моделей, хранящихся в свойствах объекта задания, в процессе сериализации самого задания.

Следует отметить, что сериализуется на все содержимое объекта модели, а только ключ представляемой этим объектом записи. Впоследствии, при десериализации модели фреймворк ищет в таблице запись с сохраненным ключом и создает на ее основе новый объект модели. Такой подход позволяет уменьшить нагрузку на подсистему очередей.

Если в свойствах задания не предполагается хранить модели, трейт можно удалить.

В классе отложенного задания должны присутствовать два метода:

- конструктор — изначально «пустой». Может использоваться для получения каких-либо значений, необходимых для выполнения задания;
- `handle()` — должен выполнять действия, ради которых и создается отложенное задание. Не обязан принимать параметры, однако может использоваться для получения каких-либо нужных для работы значений посредством внедрения зависимостей. Также не должен возвращать результат.

В классе отложенного задания можно объявить свойства для хранения нужных для работы значений, в том числе и объектов моделей. Надо при этом помнить, что сериализуются и сохраняются в очереди только значения общедоступных и защищенных свойств. Еще можно объявить произвольные методы.

В листинге 25.1 показан код отложенного задания `App\Jobs\BbDelete`, принимающего через параметр конструктора объявление в виде объекта модели `Bb` и удаляющего само объявление и связанную с ним иллюстрацию, которая хранится в отдельном файле.

Листинг 25.1. Код отложенного задания `App\Jobs\BbDelete`

```
namespace App\Jobs;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldBeUnique;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Illuminate\Support\Facades\Storage;
class BbDelete implements ShouldQueue {
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    protected $bb;

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

    public function handle() {
        if ($this->bb->pic)
            Storage::disk('public')->delete($this->bb->pic);
        $this->bb->delete();
    }
}
```

Запустить это задание на выполнение можно с помощью следующего выражения:

```
use App\Jobs\BbDelete;
. . .
BbDelete::dispatch($bb);
```

Если в свойстве отложенного задания был сохранен объект первичной модели с множеством связанных записей вторичной модели, при десериализации этой модели производится загрузка всех связанных записей, в результате чего воссоздаваемый объект модели отнимет много памяти. Если связанные записи для выполнения задания не требуются, можно сэкономить память, отдав указание не загружать эти записи, вызвав у объекта модели метод `withoutRelations()`. Пример:

```
class BbDelete implements ShouldQueue {
    . . .
    protected $bb;

    public function __construct($bb) {
        $this->bb = $bb->withoutRelations();
    }
    . . .
}
```

25.2.1.2. Параметры отложенных заданий-классов

Можно указать дополнительные параметры отложенного задания, объявив в его классе особые свойства и методы. Заданные в них значения будут перекрывать значения, заданные при запуске обработчика отложенных заданий (см. далее). Вот эти свойства и методы:

- `timeout` — свойство, общедоступное, задает максимальный промежуток времени, в течение которого обработчик отложенных заданий ждет завершения выполнения текущего задания, в секундах. По истечении этого времени дочерний процесс, выполняющий задание, будет остановлен. Это делается для того, чтобы «зависшие» задания не отнимали системные ресурсы.

  Значение свойства `timeout` должно быть чуть меньше значения рабочей настройки `retry_after` (см. *разд. 25.1.1*). Тогда сначала будет остановлено выполнение «зависшего» задания, а потом оно будет вновь помещено в очередь, после чего обработчик попытается выполнить его еще раз.

  Если свойство не указано или хранит `null`, используется значение промежутка времени, заданное при запуске обработчика отложенных заданий;

- `failOnTimeout` — свойство, общедоступное. Если `True`, то по истечении промежутка времени, заданного свойством `timeout`, задание будет помечено как проваленное. Если `false` — задание будет возвращено в список отложенных заданий, и впоследствии будут предприняты попытки снова выполнить его. По умолчанию: `false`;

- `tries` — свойство, общедоступное, задает количество попыток выполнения текущего задания. Если после всех попыток задание выполнить не удалось (что может случиться, например, вследствие постоянно возникающей ошибки), оно будет помещено в список проваленных заданий. Если свойство не указано или хранит `null`,

Laravel использует значение количества попыток, заданное при запуске обработчика отложенных заданий (см. *разд. 25.6.1*);

- `retryUntil()` — метод, общедоступный, должен возвращать время, до которого фреймворку следует пытаться выполнить текущее задание. Это время можно задать в виде объекта класса `Carbon` или любом формате, поддерживаемом PHP. Пример:

```
class BbDeleteJob implements ShouldQueue {
    . . .
    public function retryUntil() {
        return now()->addSeconds(5);
    }
}
```

Можно объявить в классе задания либо свойство `tries`...

...либо метод `retryUntil()`.

- `backoff` — свойство, общедоступное, задает промежуток времени между попытками выполнить текущее задание в секундах. Если не указано или хранит `null`, Laravel использует значение промежутка времени, заданное при запуске обработчика отложенных заданий;

- `backoff()` — метод, общедоступный, должен возвращать промежуток времени между попытками выполнить текущее задание в секундах.

  Из метода также можно вернуть массив, содержащий значения промежутков времени в секундах. Размер этого массива задаст количество попыток. Пример:

```
class BbDeleteJob implements ShouldQueue {
    . . .
    public function backoff() {
        // Будут предприняты три попытки выполнять текущее
        // задание:
        // первая — спустя 1 секунду, вторая — спустя 5 секунд,
        // третья — спустя 15 секунд
        return [1, 5, 15];
    }
}
```

  Если метод не объявлен, значение промежутка будет извлекаться из свойства `backoff` класса модели;

- `maxExceptions` — свойство, общедоступное, задает максимально допустимое количество необработанных исключений, которое может быть возбуждено при выполнении кода отложенного задания. По достижении заданного значения задание будет помещено в список проваленных заданий, даже если максимальное количество попыток его выполнения еще не исчерпано. Если свойство не указано или хранит `null`, Laravel сочтет задание проваленным после первого же исключения.

  Объекты моделей, хранящиеся в свойствах отложенного задания, перед сохранением задания в очереди сериализуются в строковое представление, а при извлечении задания из очереди — десериализуются. Однако если какой-либо из объектов модели представляет запись, уже удаленную из базы данных, будет возбуждено исклю-

чение `Illuminate\Database\Eloquent\ModelNotFoundException`, которое помешает выполнению задания. Однако это поведение можно изменить;

- `deleteWhenMissingModels` — свойство, общедоступное. Если `false`, то при десериализации объекта модели, представляющего удаленную запись, будет возбуждено исключение `ModelNotFoundException`. Если `true`, то в этом случае исключение не возбуждается, а задание считается выполненным и удаляется из очереди. По умолчанию: `false`.

Также можно указать службу очередей и очередь, в которую будет помещено запускаемое отложенное задание. Для этого достаточно в теле конструктора класса задания вызвать у его объекта следующие методы:

- `onQueue(<имя очереди>)` — помещает текущее задание в очередь с указанным *именем*:

```
class BbDelete implements ShouldQueue {
    . . .
    public function __construct($bb) {
        $this->onQueue('models');
    }
}
```

- `onConnection(<имя службы очередей>)` — помещает текущее задание в очередь, хранящуюся в службе с указанным *именем*:

```
// Помещаем задание в очередь, используемую по умолчанию,
// службы database
$this->onConnection('database');

// Помещаем задание в очередь models службы database
$this->onConnection('database')->onQueue('models');
```

25.2.1.3. Обработка ошибок в отложенных заданиях-классах

В классе отложенного задания можно объявить метод `failed()`, который будет выполнять какие-либо действия (например, отправлять соответствующее оповещение) в случае, если задание было признано фреймворком проваленным. Этот метод должен принимать в качестве параметра объект встроенного в PHP класса `Throwable`, представляющий возникшее при выполнении задания исключение, и не должен возвращать результат. Пример:

```
use Illuminate\Support\Facades\Auth;
use App\Notifications\BbDeleteFailNotification;
class BbDelete implements ShouldQueue {
    . . .
    public function failed(Throwable $exception) {
        Auth::user()->notify(new BbDeleteFailNotification);
    }
}
```

Следует отметить, что этот метод будет вызван у нового объекта задания, полученного путем повторной десериализации задания из очереди. Так что любые изменения, сделанные в свойствах объекта задания в теле метода `handle()`, к моменту вызова метода `failed()` будут утеряны.

Сообщения об ошибках, возникающих в коде отложенных заданий, на экран выводиться не будут. Для получения сведений об ошибках следует обратиться к главному журналу проекта, хранящемуся в файле storage\logs\laravel.log.

25.2.1.4. Взаимодействие с очередью

В ряде случаев в классе отложенного задания может понадобиться взаимодействовать с очередью: получить количество попыток выполнить текущее задание, вернуть его в очередь, если его не удастся выполнить в срок, или даже пометить как проваленное. Взаимодействие с очередью выполняется вызовами методов, которые добавляются в класс отложенного задания трейтом `InteractsWithQueue`:

- `attempts()` — возвращает количество попыток исполнения текущего задания;
- `release([<`*задержка*`>=0])` — вновь помещает текущее задание в очередь. Можно указать *задержку* в секундах, после которой будет предпринята следующая попытка выполнить текущее задание. Пример:

```
public function handle() {
    . . .
    if ($this->isTooLong) {
        // Выполнение задания затянется, поэтому возвращаем его
        // в очередь на 5 секунд
        $this->release(5);
    } else {
        // Выполняем задание
    }
}
```

- `fail([<`*исключение*`>=null])` — помечает задание как проваленное. Можно указать *исключение*, возбуждение которого привело к провалу задания;
- `delete()` — удаляет текущее задание из очереди.

25.2.1.5. Уникальные задания

Уникальное отложенное задание может присутствовать в очереди только в одном экземпляре. Уникальными обычно делают задания, которые нельзя запускать одновременно.

> ***Подсистема уникальных заданий…***
>
> …использует распределенные блокировки, чтобы обозначить наличие в очереди уникального задания. О распределенных блокировках рассказано в *разд. 29.1.3*.

Чтобы превратить отложенное задание в уникальное, следует дополнительно реализовать в нем один из следующих интерфейсов:

- `Illuminate\Contracts\Queue\ShouldBeUnique` — если задание должно оставаться уникальными до момента, когда оно успешно окончит выполняться или будет признано проваленным. Только после этого в очередь можно будет поместить другое задание того же типа.

  Этот интерфейс уже импортирован в модуле, где объявлен класс задания;

- `Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing` — если задание должно оставаться уникальным до момента, когда оно начнет выполняться. То есть будет можно создать новое задание того же типа, не дожидаясь окончания выполнения имеющегося задания.

Чтобы пометить существование в очереди уникального задания, Laravel создает распределенную блокировку с именем формата:

`laravel_unique_job:<`*путь к классу уникального задания*`>[<`*суффикс*`>]`

Суффикс используется в случаях, когда требуется создать несколько уникальных заданий, принадлежащих к одному и тому же классу, но выполняющих разные действия. Например, использовав в качестве *суффикса* ключ объявления, можно создать отдельные уникальные задания для обработки объявлений с ключами 1, 2 и 3, однако попытка создать еще одно уникальное задание для обработки объявления с ключом 2 потерпит неудачу.

В классе уникального задания можно объявить следующие свойства и методы:

- `uniqueId()` — метод, общедоступный, должен возвращать строковый суффикс, используемый при формировании имени распределенной блокировки (см. ранее). Если не объявлен, суффикс не будет использоваться. Пример:

```
class BbDelete implements ShouldQueue {
    . . .
    public function uniqueId() {
        return (string) $this->bb->id;
    }
}
```

- `uniqueFor` — свойство, общедоступное, хранит промежуток времени, по истечении которого текущее задание перестает быть уникальным (и появляется возможность создать новое задание того же класса), в секундах:

```
class BbDelete implements ShouldQueue, ShouldBeUnique {
    public $uniqueFor = 300;
    . . .
}
```

  Если свойство не объявлено или хранит 0, задание останется уникальным до окончания или до начала его выполнения — в зависимости от реализованного в его классе интерфейса (см. ранее);

- `uniqueFor()` — метод, общедоступный, должен возвращать промежуток времени, по истечении которого текущее задание перестает быть уникальным, в секундах. Если не объявлен, значение промежутка времени берется из свойства `uniqueFor`.

  Изначально для хранения распределенной блокировки фреймворк использует кеш по умолчанию. Однако это поведение можно изменить;

- `uniqueVia()` — метод, общедоступный, должен возвращать объект службы кеширования, в которой будет храниться распределенная блокировка:

```
use Illuminate\Support\Facades\Cache;
. . .
```

```
public function uniqueVia() {
    return Cache::store('database');
}
```

25.2.1.6. Неотложные задания

Если при подаче команды `make:job` утилиты artisan был указан командный ключ `--sync`, будет создано *неотложное задание*, выполняемое немедленно и в текущем процессе. В отличие от класса отложенного задания, класс неотложного задания:

- не реализует интерфейс `ShouldQueue`;
- не включает трейтов `InteractsWithQueue`, `Queueable` и `SerializesModels`, поскольку неотложное задание не заносится в очередь.

В виде неотложных заданий обычно реализуются операции, выполняемые в разных действиях контроллеров и не требующие много времени на исполнение.

25.2.2. Запуск отложенных заданий-классов

Для запуска отложенных заданий применяются статические методы, добавляемые в класс отложенного задания трейтом `Dispatchable`:

- `dispatch([<`*параметр 1*`>, <`*параметр 2*`>, ... <`*параметр n*`>])` — запускает отложенное задание, представляемое текущим классом. Можно указать *параметры*, которые будут переданы конструктору класса задания при его вызове. Пример:

  ```
  BbDelete::dispatch($bb);
  ```

- `dispatchIf()` — аналогичен `dispatch()`, только запускает отложенное задание лишь в том случае, если заданное *условие* в результате вычисления дает `true`:

  `dispatchIf(<`*условие*`>[, <`*параметр 1*`>, <`*параметр 2*`>, ... <`*параметр n*`>])`

  Пример:

  ```
  BbDelete::dispatchIf(request()->boolean('i_agree'), $bb);
  ```

- `dispatchUnless()` — аналогичен `dispatch()`, только запускает отложенное задание лишь в том случае, если заданное *условие* в результате вычисления дает `false`. Формат вызова такой же, как и у метода `dispatchIf()`.

  Точный момент выполнения отложенного задания, запущенного описанными ранее методами, предсказать невозможно. Задание может быть выполнено как после генерирования и отправки серверного ответа, так и до этого. В некоторых случаях это может быть критично;

- `dispatchAfterResponse()` — аналогичен `dispatch()`, только указывает фреймворку выполнить текущее отложенное задание лишь после отправки серверного ответа. Формат вызова такой же, как и у метода `dispatch()`. Пример:

  ```
  SendMail::dispatchAfterResponse(request()->user());
  ```

  Посредством описанных здесь методов также запускаются неотложные задания;

- `dispatchSync()` — аналогичен `dispatch()`, только запускает текущее задание для немедленного выполнения, как будто оно является неотложным. Формат вызова такой же, как и у метода `dispatch()`.

В предыдущих версиях Laravel для этой цели использовался метод `dispatchNow()`. Он поддерживается до сих пор для совместимости, однако подлежит удалению в будущих версиях фреймворка.

По умолчанию запускаемые задания (если они не были созданы методом `dispatchSync()`) помещаются в очередь по умолчанию, хранящуюся в службе, которая помечена как используемая по умолчанию.

У результата, возвращенного описанными здесь методами, можно вызвать приведенные далее методы, задающие дополнительные параметры задания:

- `afterCommit()` — указывает выполнить текущее задание только после успешного подтверждения всех транзакций во всех используемых базах данных. Применяется, если рабочая настройка `after_commit` хранит значение `false` (см. *разд. 25.1.1*). Пример:

```
BbDelete::dispatch($bb)->afterCommit();
```

- `beforeCommit()` — указывает, что текущее задание может быть выполнено в произвольный момент времени, в том числе и до успешного завершения всех транзакций в базах данных. Применяется, если рабочей настройке `after_commit` дано значение `true`;
- `delay(<временна́я отметка>)` — указывает выполнить задание по достижении заданной *временно́й отметки*. Последнюю можно задать в виде целого числа в секундах, значения типа `DateInterval`, `DateTimeInterface` или `Carbon`. Пример:

```
SendMail::dispatch($user)->delay(now()->addMinutes(3));
```

- `onQueue(<имя очереди>)` — помещает текущее задание в очередь с указанным *именем*:

```
BbDelete::dispatch($bb)->onQueue('models');
```

- `onConnection(<имя службы очередей>)` — помещает текущее задание в очередь, хранящуюся в службе с указанным *именем*:

```
// Помещаем задание в очередь, используемую по умолчанию, службы
// database
SendMail::dispatch($user)->onConnection('database');

// Помещаем задание в очередь mail службы database
SendMail::dispatch($user)->onConnection('database')
                         ->onQueue('mail');
```

25.3. Отложенные задания-функции

Отложенное задание также можно оформить в виде функции (*задание-функция*). Обычно в таком виде оформляются простые задания, выполняемые только в одном месте кода сайта.

Задание-функция создается и запускается вызовом функции-хелпера `dispatch(<анонимная функция>)`. Заданная *анонимная функция*, собственно и реализующая задание, не должна принимать параметров и возвращать результат. Пример:

```
dispatch(function () {
    Mail::to($user)->send(new AlertMail($user->name));
});
```

У объекта, возвращаемого функцией `dispatch()`, можно вызвать методы: `afterCommit()`, `beforeCommit()`, `onConnection()`, `onQueue()` и `delay()`, описанные в *разд. 25.2.2*. Также поддерживается метод `afterResponse()`, указывающий выполнить задание только после отправки серверного ответа. Пример:

```
dispatch(function () {
    Mail::to($user)->send(new AlertMail($user->name));
})->onConnection('database')->onQueue('mail')->afterResponse();
```

Еще этот объект поддерживает метод `catch(<`*анонимная функция*`>)`, задающий *анонимную функцию*, которая будет выполнена при возникновении исключения в коде задания. Эта *функция* в качестве параметра должна принимать объект возникшего исключения. Пример:

```
dispatch(function () {
    Mail::to($user)->send(new AlertMail($user->name));
})->catch(function ($e) {
    . . .
});
```

25.4. Цепочки отложенных заданий

Иногда бывает необходимо выполнить какое-либо отложенное задание (назовем его *ведомым*) или целую их группу только после того, как будет успешно выполнено другое задание (*ведущее*). Для этого достаточно объединить задания в *цепочку*, поставив в ее начале ведущее задание, а за ним — ведомые.

Создание цепочки заданий на основе заданного *массива* выполняется вызовом метода `chain(<`*массив заданий*`>)` фасада `Illuminate\Support\Facades\Bus`. Ведущее задание должно быть указано в *массиве* первым элементом. В *массиве* могут присутствовать как объекты заданий-классов, так и задания-функции.

Для запуска созданной цепочки заданий следует вызвать у объекта, возвращенного методом `chain()`, метод `dispatch()`.

Пример:

```
use Illuminate\Support\Facades\Bus;
. . .
Bus::chain([new BbDeletePrepare, new BbDeleteFiles($bb),
            function () use ($bb) { ... },
            new BbDeleteRecord($bb)])
   ->dispatch();
```

Также создать цепочку можно вызовом у класса задания, которое станет ведущим, метода `withChain(<`*массив ведомых заданий*`>)`. Пример:

```
BbDeletePrepare::withChain([new BbDeleteFiles($bb),
                            function () use ($bb) { ... },
                            new BbDeleteRecord($bb)])
                ->dispatch();
```

К сожалению, при создании цепочки заданий таким образом передать конструктору класса ведущего задания какие-либо параметры невозможно. Поэтому конструктор ведущего задания не должен принимать параметры, по крайней мере, обязательные.

В качестве ведущего можно использовать задание-функцию, вызвав у нее метод `chain()`, имеющий тот же формат вызова, что и метод `withChain()`:

```
dispatch(function (){
    . . .
})->chain([ ... ])->dispatch();
```

Объект цепочки заданий поддерживает методы `afterCommit()`, `beforeCommit()`, `onConnection()`, `onQueue()` и `delay()`, описанные в *разд. 25.2.2*. Пример:

```
BbDeletePrepare::withChain([ ... ])->onConnection('database')
                                   ->onQueue('models')
                                   ->dispatch();
```

25.5. Специфические разновидности отложенных заданий

25.5.1. Отложенные слушатели событий

Отложенный слушатель при возникновении события записывается в очередь и выполняется в параллельном процессе (слушатели были описаны в *разд. 22.1.1*). В виде отложенных слушателей рекомендуется оформлять обработчики событий, выполняющиеся в течение длительного времени.

Отложенный слушатель должен реализовывать интерфейс `Illuminate\Contracts\Queue\ShouldQueue`. В новом классе слушателя, сгенерированном командой `make:listener` утилиты artisan, этот интерфейс уже импортирован, так что его останется лишь добавить в объявление класса слушателя:

```
use Illuminate\Contracts\Queue\ShouldQueue;
. . .
class UserAttemptListener implements ShouldQueue {
    . . .
}
```

В остальном класс отложенного слушателя ничем не отличается от класса обычного.

По умолчанию отложенный слушатель заносится в очередь по умолчанию, хранящуюся в службе, которая отмечена как используемая по умолчанию, и выполняется без всяких задержек. Однако это поведение можно изменить, объявив в классе слушателя следующие свойства и методы:

- `afterCommit` — свойство, общедоступное. Если `true`, слушатель будет выполнен после подтверждения всех транзакций во всех базах данных. Если `false`, слушатель может быть выполнен в любой момент времени. По умолчанию: `false`;
- `queue` — свойство, общедоступное, задает имя очереди, куда будет помещен текущий слушатель;

- `viaQueue()` — метод, общедоступный, должен возвращать имя очереди, куда будет помещен текущий слушатель.

  Следует объявить либо свойство `queue`, либо метод `viaQueue()`;

- `connection` — свойство, общедоступное, задает имя службы очередей, где будет храниться текущий слушатель;

- `viaConnection()` — метод, общедоступный, должен возвращать имя службы очередей, где будет храниться текущий слушатель.

  Следует объявить либо свойство `connection`, либо метод `viaConnection()`;

- `delay` — свойство, общедоступное, задает задержку перед выполнением отложенного слушателя в секундах;

- `shouldQueue()` — метод, общедоступный. В качестве параметра должен принимать объект события. Если вернет `true`, текущий слушатель будет выполнен, если `false` — не будет выполнен. Пример:

```
class BbDeletedListener implements ShouldQueue {
    . . .
    public function shouldQueue(BbDeletedEvent $event) {
        return !empty($event->bb->pic);
    }
}
```

Также поддерживаются свойство `tries` и метод `retryUntil()`, описанные в *разд. 25.2.1.2*.

Чтобы получить возможность взаимодействовать с очередью из слушателя, следует добавить в его класс трейт `InteractsWithQueue`:

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
. . .
class UserAttemptListener implements ShouldQueue {
    use InteractsWithQueue;
    . . .
}
```

В классе слушателя можно объявить метод `failed()`, выполняемый, если слушатель был признан фреймворком проваленным. Этот метод должен принимать в качестве параметров объект события и объект исключения, возникшего при выполнении слушателя. Пример:

```
class UserAttemptListener implements ShouldQueue {
    . . .
    public function failed(Attempting $event, $exception) {
        . . .
    }
}
```

Слушатели-функции также можно сделать отложенными. Для этого достаточно анонимную функцию, передаваемую методу `listen()` фасада `Event`, «обернуть» в функцию `Illuminate\Events\queueable()`. Пример:

```
use function Illuminate\Events\queueable;
use \Illuminate\Auth\Events\Attempting;
. . .
Event::listen(queueable(
                    function (Attempting $event) {
                        . . .
                    }
                )
);
```

Объект отложенного слушателя, возвращаемый функцией `queueable()`, поддерживает методы: `onQueue()`, `onConnection()` и `delay()`, задающие очередь, службу очередей и задержку соответственно и описанные в *разд. 25.2.2*. Пример их использования:

```
Event::listen(queueable(
                    function (Attempting $event) {
                        . . .
                    }
                )
     ->onQueue('mail')->onConnection('database'));
```

Также этот объект поддерживает метод `catch(<анонимная функция>)`, задающий *анонимную функцию*, которая будет выполнена при возникновении исключения в коде обработчика. Эта *функция* в качестве параметров должна принимать объект обрабатываемого события и объект возникшего исключения. Пример:

```
Event::listen(queueable(
                    function (Attempting $event) {
                        . . .
                    }
                )->catch(
                    function (Attempting $event, $e) {
                        . . .
                    }
                )
);
```

25.5.2. Отложенные электронные письма

Отложенное электронное письмо при отправке сохраняется в очереди, а его непосредственная отправка выполняется в отдельном процессе. Если сайт отправляет множество электронных писем, имеет смысл превратить их в отложенные, чтобы повысить отзывчивость сайта.

Любое электронное письмо (подробности об отправке электронной почты — в *главе 23*) может быть превращено в отложенное. Для этого необходимо:

- добавить в объявление его класса интерфейс `ShouldQueue`.

  В новом классе электронного письма, сгенерированного командой `make:mail` утилиты artisan, этот интерфейс уже импортирован;

- добавить в его класс трейты `Queueable` и `SerializesModels` (последний — только если класс письма содержит свойства, хранящие объекты моделей). Впрочем, вновь созданный класс письма уже включает их.

Пример:

```
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
class SimpleMail extends Mailable implements ShouldQueue {
    use Queueable, SerializesModels;
    . . .
}
```

После чего при отправке любого письма, представленного объектом такого класса, вызовом метода `send()` оно будет отсылаться как отложенное.

Обычное электронное письмо также можно отправить как отложенное, использовав вместо метода `send()` метод `queue(<`*объект письма*`>[, <`*имя очереди*`>=null])`. Если *имя очереди* не указано, будет использована очередь по умолчанию в службе, помеченной как используемая по умолчанию. Пример:

```
use Illuminate\Support\Facades\Mail;
. . .
Mail::to('user@bboard.ru')->queue(new SimpleMail('user'));
```

Также можно вызвать метод `later()`, отправляющий отложенное письмо в момент времени, заданный *временно́й отметкой*:

```
later(<временна́я отметка>, <объект письма>[, <имя очереди>=null])
```

Временну́ю отметку можно задать в виде значения типа `DateInterval`, `DateTimeInterface` или `Carbon`. Пример:

```
Mail::to('user@bboard.ru')
    ->later(now()->addMinutes(3), new SimpleMail('user'));
```

Трейт `Queueable` добавляет классу электронного письма поддержку методов: `afterCommit()`, `beforeCommit()`, `onQueue()`, `onConnection()` и `delay()`, описанных в *разд. 25.2.2*. Эти методы могут быть вызваны как у созданного объекта письма…

```
$mail = new SimpleMail('user')->onQueue('mail')
                              ->onConnection('database');
Mail::to('user@bboard.ru')->queue($mail);
```

…так и в конструкторе класса письма:

```
class SimpleMail extends Mailable implements ShouldQueue {
    . . .
    public function __construct() {
        $this->onQueue('mail');
        $this->onConnection('database');
    }
    . . .
}
```

25.5.3. Отложенные оповещения

Отложенные оповещения при отправке также сохраняются в очереди, а собственно их отсылка выполняется в отдельном процессе (оповещения описывались в *главе 24*). Превратить оповещение в отложенное можно способом, описанным в *разд. 25.5.2* (единственное: трейт `Illuminate\Queue\SerializesModels` там изначально не импортирован). Пример:

```
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\SerializesModels;
class SimpleNotification extends Notification implements ShouldQueue
{
    use Queueable, SerializesModels;
    . . .
}
```

После чего при отправке любого оповещения, представленного объектом такого класса, вызовом метода `send()` фасада `Notification` или метода `notify()` объекта-адресата оно будет отсылаться как отложенное.

Изначально оповещения сохраняются в очереди по умолчанию, содержащейся в службе по умолчанию, и выполняются без всяких задержек. Однако это поведение можно изменить, объявив в классе оповещения следующие свойства и методы:

- `connection` — свойство, общедоступное, задает имя службы очередей, где будет храниться текущее оповещение;
- `queue` — свойство, общедоступное, задает имя очереди, куда будет помещено текущее оповещение.

  В качестве значения свойства также можно указать ассоциативный массив. Ключами его элементов станут имена каналов, по которым будет пересылаться оповещение, а значениями элементов — собственно имена очередей:

  ```
  class SimpleNotification extends Notification implements ShouldQueue {
      public $queue = ['mail' => 'mail-queue',
                       'vonage' => 'sms-queue'];
      . . .
  }
  ```

- `viaQueues()` — метод, общедоступный, должен возвращать имя очереди, куда будет помещено текущее оповещение, или массив с именами очередей.

  Следует объявить либо свойство `queue`, либо метод `viaQueues()`;

- `delay` — свойство, общедоступное, задает время отправки письма в виде временно́й отметки.

  В качестве значения свойства также можно указать ассоциативный массив. Ключами его элементов станут имена каналов, по которым будет пересылаться оповещение, а значениями элементов — значения времени отправки оповещений:

  ```
  class SimpleNotification extends Notification implements
                                                  ShouldQueue {
  ```

```
    public $delay = ['mail' => now()->addMinutes(3),
                     'vonage' => now()->addMinutes(5)];
    . . .
}
```

- `shouldQueue()` — метод, общедоступный. В качестве параметра должен принимать объект-адресат и имя канала. Если вернет `true`, текущее оповещение будет отправлено, если `false` — не будет отправлено. Пример:

```
class SimpleNotification extends Notification implements
                                                ShouldQueue {
    . . .
    public function shouldQueue($notifiable, $channel) {
        return ($channel == 'vonage') ?
                    ($notifiable->name == 'admin') : true;
    }
}
```

Можно выполнить немедленную отправку отложенного оповещения — одним из следующих способов:

- вызвав метод `sendNow()` фасада `Notification`, имеющего тот же формат вызова, что и метод `send()`:

```
use Illuminate\Support\Facades\Notification:
. . .
Notification::sendNow(Auth::user(), new FailureNotification);
```

- если класс объекта-адресата содержит трейт `Notifiable` — вызвав у объекта-адресата метод `notifyNow()` в том же формате, что и метод `notify()`:

```
Auth::user()->notifyNow(new SimpleNotification);
```

Трейт `Queueable` добавляет классу оповещения поддержку методов: `afterCommit()`, `beforeCommit()`, `onQueue()`, `onConnection()` и `delay()`, описанных в *разд. 25.2.2*. Эти методы могут быть вызваны как у созданного объекта оповещения, так и в конструкторе его класса.

25.6. Выполнение отложенных заданий

Отложенные задания, включая проваленные, выполняются специальными обработчиками, работающими в отдельных процессах. Их необходимо запускать вручную в других экземплярах командной строки.

25.6.1. Запуск обработчика отложенных заданий

Обработчик отложенных заданий запускается подачей команды:

```
php artisan queue:listen [<имя службы очередей>] ↳
[--queue=<имя очереди>] ↳
[--timeout=<максимальное время выполнения задания>] ↳
[--tries=<количество попыток обработки задания>] ↳
[--backoff=<промежуток времени между попытками выполнить задание>] ↳
```

```
[--sleep=<время «сна»>] [--memory=<занимаемый объем памяти>] ↳
[--force]
```

Будучи выполненной без ключей:

```
php artisan queue:listen
```

команда запустит обработчик, который станет просматривать очередь по умолчанию, находящуюся в службе по умолчанию.

Если указать *имя службы очередей*, обработчик будет искать задания в очереди по умолчанию из этой службы:

```
php artisan queue:listen database
```

В ключе `--queue` можно задать имя очереди:

```
php artisan queue:listen database --queue=mail
```

Также можно указать в этом ключе несколько имен очередей через запятую:

```
php artisan queue:listen database --queue=mail,notifications
```

Остальные командные ключи:

- `--timeout` — задает *максимальное время выполнения задания* в секундах (по умолчанию: 60);
- `--tries` — задает *количество попыток обработки задания* перед признанием его проваленным (по умолчанию: 1);
- `--backoff` — задает *промежуток времени между попытками выполнить* не выполняющееся вследствие ошибки *задание* в секундах (по умолчанию: 0);
- `--sleep` — задает *время*, в течение которого обработчик будет ожидать появления в очереди нового задания, если очередь «пуста», в секундах (по умолчанию: 3);
- `--memory` — задает максимальный *объем памяти*, отнимаемый обработчиком заданий, в Мбайт (по умолчанию: 128). Если требуется выполнять задания, отнимающие много системных ресурсов, возможно, придется увеличить этот объем;
- `--force` — принудительно запускает обработчик, даже если сайт находится в режиме обслуживания (будет описан в *главе 35*). Если не указан, обработчик не будет запускаться, когда сайт переведен в режим обслуживания.

Обработчик отложенных заданий будет работать постоянно. Чтобы остановить его, достаточно нажать комбинацию клавиш <Ctrl>+<C> или <Ctrl>+<Break>.

Обработчик, запущенный командой `queue:listen`, отслеживает изменение модулей с программным кодом сайта и в этом случае автоматически перезапускается. Поэтому такую команду удобно использовать при отладке.

Обработчик можно запустить и другой командой:

```
php artisan queue:work [<имя службы очередей>] ↳
[--queue=<имя очереди>] [--once] [--stop-when-empty] ↳
[--timeout=<максимальное время выполнения задания>] ↳
[--tries=<количество попыток обработки задания>] ↳
[--backoff=<промежуток времени между попытками выполнить задание>]↳
[--rest=<промежуток времени между выполнением отдельных заданий>] ↳
```

```
[--max-jobs=<количество выполняемых заданий>] ↳
[--max-time=<время работы>][--sleep=<время «сна»>]  ↳
[--memory=<занимаемый объем памяти>] [--force]
```

Она поддерживает те же самые командные ключи, что и `queue:listen`, и два дополнительных:

- `--rest` — задает *промежуток времени между выполнением отдельных заданий* в секундах (по умолчанию: 0);
- `--once` — указывает обработчику выполнить всего одно задание из очереди и тотчас завершить работу;
- `--stop-when-empty` — указывает обработчику завершить работу, если очередь «опустела»;
- `--max-jobs` — указывает обработчику выполнить заданное *количество заданий* и завершиться;
- `--max-time` — указывает обработчику отработать заданное *время* (в секундах) и завершиться.

Обработчик, запущенный командой `queue:work`, не отслеживает правку исходного кода сайта и не перезапускается после этого. Поэтому его придется перезапускать вручную. Удобнее всего использовать для этого команду:

```
php artisan queue:restart
```

Получив ее, Laravel позволит обработчику закончить выполнение текущего задания и корректно перезапустит его.

Обработчик, запускаемый командой `queue:work`, обеспечивает более высокую производительность. Поэтому ее рекомендуется использовать при эксплуатации сайта.

25.6.2. Работа с проваленными заданиями

Утилита artisan позволяет произвести с проваленными заданиями такие действия:

- вывести их список — командой:

  ```
  php artisan queue:failed
  ```

  Список проваленных заданий выводится в виде таблицы с колонками: **ID** (универсальный уникальный идентификатор), **Connection** (имя службы очередей), **Queue** (имя очереди), **Class** (путь к классу задания) и **Failed At** (временна́я отметка момента, когда задание было помечено как проваленное);

- попытаться выполнить задания с указанными *обозначениями* — командой:

  ```
  php artisan queue:retry [<обозначения заданий>] ↳
  [--queue=<имя очереди>]
  ```

  В качестве *обозначений заданий* можно указать:

  - универсальный уникальный идентификатор (выводится в колонке **ID** списка, отображаемого командой `queue:failed`):

    ```
    php artisan queue:retry 49a473d6-198e-45df-a17e-4e885e7875fc
    ```

- последовательность идентификаторов, разделенных пробелами:

```
php artisan queue:retry ↳
49a473d6-198e-45df-a17e-4e885e7875fc ↳
91401d2c-0784-4f43-824c-34f94a33c24d
```

- ключа `all` — для выполнения всех заданий:

```
php artisan queue:retry all
```

Также можно выполнить все задания в очереди с заданным *именем*, записав его в командном ключе `--queue`:

```
php artisan queue:retry --queue=models
```

❒ удалить задание с заданным *идентификатором* — командой:

```
php artisan queue:forget <идентификатор задания>
```

❒ удалить проваленные задания — командой:

```
php artisan queue:flush [--hours=<промежуток времени>]
```

По умолчанию удаляются все проваленные задания. Однако можно указать командный ключ `--hours` — и задания, созданные в течение заданного *промежутка времени* (в часах), будут сохранены. Пример удаления всех проваленных заданий, кроме созданных в течение последнего часа:

```
php artisan queue:flush --hours=1
```

Удалить проваленные задания можно и другой командой:

```
php artisan queue:prune-failed [--hours=<промежуток времени>]
```

От команды `queue:flush` она отличается тем, что, будучи запущенной без ключа `--hours`, удаляет все задания, кроме оставленных в последние 24 часа.

25.6.3. Очистка очереди заданий

Для очистки очереди заданий предназначена команда:

```
php artisan queue:clear [<имя службы очередей>] ↳
[--queue=<имя очереди>] [--force]
```

Будучи запущенной без ключей, она удалит все задания из очереди по умолчанию, находящейся в службе очередей по умолчанию. Чтобы удалить задания из очереди по умолчанию, находящейся в другой службе, следует указать *имя* этой службы. В ключе `--queue` можно задать *имя очереди*.

Ключ `--force` предписывает удалить задания, даже если сайт работает в эксплуатационном режиме.

25.7. Пакеты отложенных заданий

Пакет отложенных заданий похож на цепочку (см. *разд. 25.4*) за двумя исключениями. Во-первых, задания, объединенные в пакет, могут выполняться параллельно (в цепочке задания обрабатываются строго последовательно, друг за другом). Во-вторых, любое

задание, входящее в состав пакета, может управлять пакетом: добавлять в него новые задания, прерывать его исполнение и др.

25.7.1. Подготовка таблицы для хранения пакетов отложенных заданий

Пакеты отложенных заданий сохраняются в особой таблице базы данных, которую требуется создать.

Для формирования миграции, которая создаст эту таблицу, следует подать команду:

```
php artisan queue:batches-table
```

Создаваемая миграция получит имя формата *<временна́я отметка>*_create_job_batches_table.php. Она создает таблицу `job_batches` в базе данных по умолчанию.

После этого следует выполнить миграции.

25.7.2. Создание отложенных заданий, пригодных для использования в пакетах

Отложенное задание, пригодное для использования в составе пакетов, создается так же, как и обычное. Единственное исключение: в класс отложенного задания следует добавить трейт `Illuminate\Bus\Batchable`. Пример:

```
use Illuminate\Bus\Batchable;
class BbDelete implements ShouldQueue {
    use Dispatchable, InteractsWithQueue, Queueable,
                      SerializesModels, Batchable;
    . . .
}
```

Если предполагается, что какое-либо из отложенных заданий, входящих в состав пакета, может прервать исполнение пакета, в теле метода `handle()` каждого класса задания следует предусмотреть проверку, не было ли прервано исполнение текущего пакета. Как реализовать такую проверку, будет рассказано далее.

25.7.3. Создание и запуск пакетов отложенных заданий

Для создания пакета на основе указанных заданий применяется метод `batch(<`*массив заданий*`>)` фасада `Illuminate\Support\Facades\Bus`. В качестве элементов *массива заданий* можно указать:

- объекты заданий:

```
use Illuminate\Support\Facades\Bus;
. . .
$batch1 = Bus::batch([new BbDelete{$bb1), new BbDelete{$bb2),
                                         new BbDelete{$bb3)]);
```

- массивы объектов заданий — каждый такой массив создаст отдельную цепочку, выполняемую в составе пакета:

```
$batch2 = Bus::batch([
    new BbDelete{$bb1),
    [new BbDeletePrepare, new BbDeleteFiles($bb),
     new BbDeleteRecord($bb)],
    new BbDelete{$bb2),
    new BbDelete{$bb3)
]);
```

Метод `batch()` возвращает объект класса `Illuminate\Bus\PendingBatch`, который представляет *еще не выполнившийся* пакет. Чтобы запустить пакет на выполнение, достаточно вызвать у него один из двух следующих методов:

- `dispatch()` — запускает пакет на выполнение. При этом выполнение пакета может начаться в произвольный момент времени. Пример:

  ```
  $batch1->dispatch();
  ```

- `dispatchAfterResponse()` — запускает пакет на выполнение, которое начнется строго после отправки серверного ответа:

  ```
  $batch2->dispatchAfterResponse();
  ```

Изначально созданный таким образом пакет будет помещать входящие в него задания в очередь по умолчанию, хранящуюся в службе по умолчанию. Если какое-либо задание окажется невыполненным и будет признано фреймворком проваленным, выполнение пакета прервется.

Класс `PendingBatch` поддерживает ряд полезных методов, которые задают дополнительные настройки и должны быть вызваны перед запуском пакета на выполнение:

- `add(<`*массив заданий*`>)` — добавляет в текущий пакет задания из указанного *массива*;
- `then(<`*анонимная функция*`>)` — задает *анонимную функцию*, которая будет вызвана после успешного выполнения всех заданий из текущего пакета. *Функция* должна принимать в качестве параметра объект класса `Illuminate\Bus\Batch`, представляющий *выполняющийся или уже выполнившийся* пакет, и не возвращать результата;
- `catch(<`*анонимная функция*`>)` — задает *анонимную функцию*, которая будет вызвана в случае возникновения исключения в каком-либо из заданий, входящих в текущий пакет. *Функция* должна принимать два параметра: объект класса `Batch`, представляющий пакет, и объект возникшего исключения. Результат она возвращать не должна;
- `finally(<`*анонимная функция*`>)` — задает *анонимную функцию*, которая будет выполнена после завершения выполнения пакета, неважно, успешного или неуспешного. *Функция* должна принимать в качестве параметра объект класса `Batch`, представляющий пакет, и не возвращать результат.

  Пример:

  ```
  Bus::batch([ ... ])
     ->then(function ($batch) {
           // Пакет успешно выполнился
       })
     ->catch(function ($batch, $exception) {
           // Пакет выполнился неуспешно. Где-то возникла ошибка.
       })
  ```

```
    ->finally( function ($batch) {
          // Выполнение пакета завершено, успешно или неуспешно
      })
    ->dispatch();
```

- `allowFailures()` — указывает, продолжать ли выполнение текущего пакета после возникновения первого исключения. Формат вызова:

  ```
  allowFailures([<продолжать выполнение пакета?>=true])
  ```

  Если параметру *продолжать выполнение пакета* дано значение `false`, после первого исключения, возникшего в каком-либо из заданий, входящих в состав пакета, выполнение пакета прервется. Если параметру дано значение `true`, выполнение пакета продолжится;

- `name(<имя>)` — дает текущему пакету указанное *имя*. Именование пакетов может пригодиться при отладке.

Также поддерживаются методы `onConnection()` и `onQueue()`, описанные в *разд. 25.2.2*.

25.7.4. Взаимодействие с выполняющимся пакетом

Имеется возможность взаимодействия с выполняющимся пакетом из кода любого из входящих в него заданий. В частности, можно добавлять в пакет новые задания, прерывать выполнение пакета, проверять, было ли прервано его исполнение одним из ранее выполнившихся заданий, и др.

Выполняющийся и выполнившийся пакет представляется объектом класса `Batch`. Этот объект в коде задания, входящего в состав пакета, можно получить вызовом метода `batch()`, который добавляется классу задания с трейтом `Batchable`.

Класс `Batch` поддерживает следующие методы:

- `add(<массив заданий>)` — добавляет в текущий пакет задания из указанного *массива*:

  ```
  class FillBatch implements ShouldQueue {
      . . .
      public function handle() {
          $this->batch()->add([new DoSomeJob, new DoAnotherJob]);
      }
  }
  ```

- `cancel()` — прерывает выполнение текущего пакета;

- `cancelled()` — возвращает `true`, если исполнение текущего пакета было прервано, и `false` — в противном случае:

  ```
  public function handle() {
      // Сначала проверяем, было ли исполнение пакета прервано
      // каким-либо из предыдущих заданий
      if ($this->batch()->cancelled())
          // Если исполнение было прервано, завершаем
          // выполнение задания
          return;
      // В противном случае выполняем задание
      . . .
  ```

```
        // При небходимости прерываем исполнение пакета
        if ($this->thisBatchMustBeCancelled())
            $this->batch()->cancel();
    }
```

- `canceled()` — то же самое, что и `cancelled()`.

Также поддерживается метод `allowFailures()` (см. *разд. 25.7.3*).

25.7.5. Получение сведений о пакете

Для получения разнообразных сведений о выполняющемся или выполнившемся пакете класс `Batch` поддерживает ряд свойств и методов. Начнем со свойств:

- `id` — универсальный уникальный идентификатор текущего пакета;
- `name` — имя текущего пакета или «пустая» строка, если таковое не было задано;
- `totalJobs` — общее количество заданий в текущем пакете;
- `pendingJobs` — количество еще не выполненных заданий;
- `failedJobs` — количество проваленных заданий;
- `createdAt` — временна́я отметка создания текущего пакета в виде объекта класса `Carbon`;
- `finishedAt` — временна́я отметка успешного завершения текущего пакета в виде объекта класса `Carbon` или `null`, если пакет еще исполняется или завершился неуспешно;
- `cancelledAt` — временна́я отметка неуспешного завершения текущего пакета в виде объекта класса `Carbon` или `null`, если пакет еще исполняется или завершился успешно.

Теперь рассмотрим методы:

- `processedJobs()` — возвращает количество выполненных заданий;
- `progress()` — возвращает процент выполненных заданий в виде числа от 0 до 100;
- `finished()` — возвращает `true`, если пакет завершился успешно, и `false` — в противном случае;
- `cancelled()` — возвращает `true`, если пакет завершился неуспешно, и `false` — в противном случае;
- `canceled()` — то же самое, что и `cancelled()`.

Если известен идентификатор пакета (получаемый из свойства `id`), соответствующий пакет можно найти с помощью метода `findBatch(<идентификатор пакета>)` фасада `Bus`. Метод возвращает объект класса `Batch`, представляющий пакет с указанным идентификатором, или `null`, если такой пакет не существует.

Объект класса `Batch` можно вернуть из действия контроллера в качестве результата. В таком случае этот объект будет преобразован в объект JSON со свойствами `id`, `name`, `totalJobs`, `pendingJobs`, `failedJobs`, `createdAt`, `finishedAt`, `cancelledAt`, `processedJobs` и `progress`, взятых из одноименных свойств и методов. Пример:

```
Route::get('/batch/{batchId}',
           [AdminController::class, 'getBatch']);
. . .
use Illuminate\Support\Facades\Bus;
class AdminController extends Controller {
    . . .
    public function getBatch($batchId) {
        return Bus::findBatch($batchId);
    }
}
```

Это может пригодиться при программировании административного раздела сайта, выдающего статистику выполнения пакетов отложенных заданий.

25.7.6. Выполнение пакетов отложенных заданий

Пакеты отложенных заданий обрабатываются тем же обработчиком, что и обычные отложенные задания (см. *разд. 25.6.1*).

Осуществить попытку повторного выполнения пакета с указанным *идентификатором*, ранее не выполненного вследствие ошибки в одном из входящих в него заданий, позволит команда:

```
php artisan queue:retry-batch <идентификатор пакета>
```

К сожалению, выполненные пакеты не удаляются из таблицы `jobs_batches` автоматически. Очистить эту таблицу от выполнившихся пакетов позволяет команда:

```
php artisan queue:prune-batches [--hours=<промежуток времени>] ↪
[--unfinished=<промежуток времени>]
```

По умолчанию будут удалены пакеты, успешно выполнившиеся не менее 24 часов назад. Пакеты, которые обработчик пытался выполнить несколько раз, и ни одна попытка выполнения которых не увенчалась успехом, удаляться не будут.

Поддерживаются следующие командные ключи:

- `--hours` — задает *промежуток времени*, который должен пройти после успешного выполнения пакетов, подлежащих удалению, в часах. Пакеты, выполнившиеся до истечения указанного *промежутка времени*, удаляться не будут. По умолчанию: 24;
- `--unfinished` — предписывает удалить все ошибочные пакеты, после последней безуспешной попытки выполнить которые прошел указанный (в часах) *промежуток времени*. Если не указан, ошибочные пакеты удаляться не будут.

Пример:

```
php artisan queue:prune-batches --hours=12 --unfinished=48
```

Эту команду рекомендуется запускать регулярно, для чего удобно использовать встроенный в Laravel планировщик заданий (см. *главу 27*).

Удалить из списка ошибочные пакеты также можно подачей команды:

```
php artisan queue:prune-failed [--hours=<промежуток времени>]
```

Командный ключ `--hours` задает *промежуток времени*, который должен пройти после последней попытки выполнить пакеты, подлежащие удалению, в часах. Пакеты, которые обработчик пытался выполнить до истечения указанного *промежутка*, удаляться не будут. По умолчанию *промежуток времени* равен 24 часам.

25.8. Посредники отложенных заданий

Посредник отложенных заданий выполняет предварительную обработку отложенного задания перед передачей его обработчику. Такие посредники могут ограничивать частоту выполнения заданий, предотвращать одновременное выполнение заданий, относящихся к определенному классу, и т. п. Посредники отложенных заданий реализованы в виде классов.

Чтобы указать посредники, которые будут действовать на задания, относящиеся к какому-либо классу, в этом классе следует объявить общедоступный метод `middleware()`. Он не должен принимать параметров и должен возвращать массив, содержащий объекты нужных посредников. Пример указания у задания посредников `RateLimited` и `ThrottleExceptions`:

```
use Illuminate\Queue\Middleware\RateLimited;
use Illuminate\Queue\Middleware\ThrottlesExceptions;
class BbDelete implements ShouldQueue {
    . . .
    public function middleware() {
        return [new RateLimited('file-deletes'),
                new ThrottlesExceptions(5, 3)];
    }
}
```

ПОСРЕДНИКИ ОТЛОЖЕННЫХ ЗАДАНИЙ…

…используют кеш проекта для хранения нужных для работы данных. О подсистеме кеширования будет рассказано в *главе 29*.

25.8.1. Ограничение частоты выполнения заданий

Ограничитель *частоты выполнения заданий* (количества заданий, выполняемых в единицу времени) создается так же, как и именованный ограничитель частоты запросов, — вызовом метода `for()` у фасада `RateLimiter`, описанного в *разд. 19.4.2*. Анонимная функция, передаваемая методу `for()`, должна принимать в качестве параметра объект задания. Код вызова этого метода также записывается в тело метода `boot()` какого-либо провайдера (обычно `AppServiceProvider`), а для создания собственно ограничителя частоты применяются те же методы.

Пример создания ограничителя частоты с именем `file-deletes`, разрешающего каждому зарегистрированному пользователю выполнять не более 10 заданий `BbDelete` в минуту:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;
```

```
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        RateLimiter::for('file-deletes', function ($job) {
            return Limit::perMinute(10)->by($job->bb->user->id);
        });
    }
}
```

Созданный таким образом ограничитель следует указать у класса задания. Он указывается в виде объекта класса `Illuminate\Queue\Middleware\RateLimited`. Конструктор этого класса вызывается в формате:

`RateLimited(<имя ограничителя частоты выполнения заданий>)`

Пример указания объекта ограничителя частоты выполнения у отложенного задания можно увидеть в *разд. 25.8*.

По умолчанию, если ограничитель не разрешает выполнить очередное задание, он возвращает его в очередь. Однако можно указать ограничителю не делать этого (не возвращенное в очередь задание никогда не будет выполнено, что может оказаться приемлемым в некоторых случаях). Для этого достаточно вызвать у объекта ограничителя метод `dontRelease()`. Пример:

```
public function middleware() {
    return [(new RateLimited('file-deletes'))->dontRelease()];
}
```

Если кеширование в проекте реализовано на основе базы данных Redis, вместо посредника `RateLimited` можно использовать посредник `Illuminate\Queue\Middleware\RateLimitedWithRedis`, специально оптимизированный для такого случая.

25.8.2. Предотвращение наложения заданий

Также имеется возможность предотвращения наложения заданий, имеющих какой-либо произвольно указанный признак. В этом случае выполнение очередного отложенного задания, имеющего этот признак, будет заблокировано, пока не завершится выполнение ранее созданного задания с тем же признаком.

> ***Посредник, предотвращающий наложение заданий…***
>
> …использует для работы распределенные блокировки (см. *разд. 29.1.3*).

Подобного рода программный инструмент можно рассматривать как аналог уникального задания (см. *разд. 25.2.1.5*), только с менее развитыми функциональными возможностями.

Посредник, предотвращающий наложение заданий, представляется классом `Illuminate\Queue\Middleware\WithoutOverlapping`. Конструктор этого класса записывается в формате:

`WithoutOverlapping(<значение признака>)`

Значение признака, на основе которого будет предотвращаться наложение заданий, может быть любого типа.

Пример указания посредника, предотвращающего одновременное выполнение заданий `BdDelete` одним и тем же пользователем:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;
class BbDelete implements ShouldQueue {
    . . .
    public function middleware() {
        return [new WithoutOverlapping($this->bb->user->id)];
    }
}
```

Если очередное извлеченное из очереди задание накладывается на уже выполняющееся, оно будет возвращено в очередь, и впоследствии будет предпринята следующая попытка его выполнить. Повторная попытка выполнить задание может быть предпринята в произвольный момент времени. Если выполняющееся задание не может быть успешно выполнено вследствие повторяющейся ошибки, оно не будет удалено из очереди, вследствие чего заблокирует выполнение всех остальных заданий с тем же признаком.

Это поведение по умолчанию, которое можно изменить с помощью следующих методов класса `WithoutOverlapping`:

- `releaseAfter(<`*временной промежуток*`>)` — задает *временной промежуток*, спустя который будет предпринята повторная попытка выполнить возвращенное в очередь задание, в секундах;
- `expireAfter(<`*временной промежуток*`>)` — задает *временной промежуток*, спустя который выполняющееся задание перестанет блокировать выполнение следующих заданий с тем же признаком, в секундах. Это может помочь избежать «вечной» блокировки выполнения последующих заданий заданием, которое не может быть выполнено вследствие повторяющейся ошибки.

  Пример:

  ```
  public function middleware() {
      return [(new WithoutOverlapping($this->bb->user->id))
                  ->releaseAfter(30)->expireAfter(60)];
  }
  ```

- `dontRelease()` — предотвращает возврат заблокированного задания в очередь, в результате чего заблокированное задание не будет выполнено никогда. Этот метод следует использовать лишь если потеря заблокированных заданий не нарушит нормальную работу сайта.

25.8.3. Отложенное выполнение ошибочных заданий

Иногда случается, что какое-либо отложенное задание не может быть выполнено вследствие постоянно возникающего исключения (что может случиться, например, при невозможности подключиться к сторонней веб-службе, которая в текущий момент времени перегружена). Тогда имеет смысл вернуть задание в очередь и попытаться выполнить снова спустя некоторое время.

Такое отложенное выполнение ошибочных заданий реализует посредник, представляемый классом `Illuminate\Queue\Middleware\ThrottlesExceptions`. Он откладывает выпол-

нение задания на заданный в минутах *временной промежуток* после того, как при выполнении этого задания возникло указанное *количество исключений*. Его конструктор вызывается в формате:

```
ThrottlesExpression([<количество исключений>=10[,
                     <временной промежуток>=5]])
```

Пример указания посредника, реализующего отложенное выполнение, можно увидеть в *разд. 25.8*.

Очередная попытка выполнить ошибочное задание, если указанное в конструкторе *количество исключений* еще не достигнуто, будет произведена без всякой задержки. А для хранения количества уже возникших исключений в задании с отложенным выполнением фреймворк создает в кеше значение с именем формата:

```
laravel_throttles_exceptions:<хеш пути к классу задания>
```

Это поведение по умолчанию, которое можно изменить с помощью следующих методов класса `ThrottlesExpression`:

- `backoff(<временной промежуток>)` — задает *временной промежуток* перед повторной попыткой выполнить задание, если указанное в конструкторе количество исключений еще не достигнуто, в минутах;
- `by(<суффикс>)` — задает *суффикс*, который будет участвовать в формировании имени значения, записываемого в кеш и хранящего количество возникших исключений. Если такой *суффикс* указан, имя этого значения будет иметь формат:

  ```
  laravel_throttles_exceptions:<суффикс>
  ```

  В результате будет подсчитываться количество возникших исключений во всех заданиях, у которых был указан один и тот же суффикс. Это пригодится, если требуется реализовать отложенное выполнение целой группы заданий (например, работающих с одной и той же сторонней веб-службой). Пример:

  ```
  use Illuminate\Queue\Middleware\ThrottlesExceptions;
  class BbDelete implements ShouldQueue {
      . . .
      public function middleware() {
          return [(new ThrottlesExceptions(5, 3))
                ->by('cloud-storage')];
      }
  }
  . . .
  class AdditionalImageDelete implements ShouldQueue {
      . . .
      public function middleware() {
          return [(new ThrottlesExceptions(5, 3))
                ->by('cloud-storage')];
      }
  }
  ```

Если кеширование в проекте реализовано на основе базы данных Redis, вместо посредника `ThrottlesExceptions` можно использовать посредник `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis`, специально оптимизированный для такого случая.

25.9. События, генерируемые при выполнении отложенных заданий

В процессе выполнения отложенных заданий генерируются следующие события, классы которых объявлены в пространстве имен `Illuminate\Queue\Events`:

- `JobQueued` — генерируется после помещения в очередь очередного задания. Поддерживаются свойства:
  - `job` — объект задания;
  - `connectionName` — имя службы очередей;
- `Looping` — генерируется перед выборкой очередного отложенного задания из очереди. Поддерживаются свойства:
  - `queue` — имя очереди;
  - `connectionName` — имя службы очередей.

  Если вернуть из обработчика значение `false`, выполнение заданий будет временно приостановлено.

  Событие генерируется только в том случае, если в очереди еще есть задания;
- `JobProcessing` — генерируется перед выполнением отложенного задания. Поддерживаются те же свойства, что и у класса `JobQueued`;
- `JobProcessed` — генерируется после успешного выполнения отложенного задания. Поддерживаются те же свойства, что и у класса `JobProcessing`;
- `JobExceptionOccurred` — генерируется, когда в коде отложенного задания возникает исключение. Поддерживаются свойства:
  - `job` — объект задания;
  - `exception` — объект возникшего исключения;
  - `connectionName` — имя службы очередей;
- `JobRetryRequested` — генерируется, когда не выполненное ранее задание вновь извлекается из очереди для выполнения. Поддерживается свойство `job`, хранящее объект задания;
- `JobFailed` — генерируется, когда отложенное задание помечается обработчиком как проваленное. Поддерживаются те же свойства, что и у класса `JobExceptionOccurred`;
- `WorkerStopping` — генерируется перед остановкой процесса-обработчика заданий. Поддерживается свойство `status`, хранящее целочисленный статус завершения процесса;
- `QueueBusy` — генерируется, если очередь переполнена. Поддерживаются свойства:
  - `queue` — имя очереди;
  - `connectionName` — имя службы очередей.
  - `size` — количество заданий в очереди.

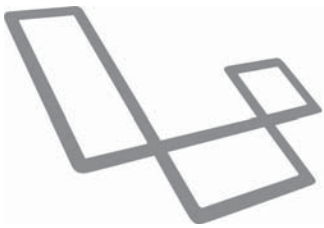
Можно привязать к этим событиям обработчики как знакомым по *главе 22* способом, так и воспользовавшись методами фасада `Illuminate\Support\Facades\Queue`. Вызовы этих методов следует располагать в каком-либо провайдере — например, `EventServiceProvider`. Вот эти методы:

- `looping(<`*`анонимная функция`*`>)` — привязывает обработчик, заданный в виде *анонимной функции*, к событию `Looping`. *Анонимная функция* должна принимать в качестве параметра объект события. Пример:

```
use Illuminate\Support\Facades\Queue;
use Illuminate\Support\Facades\DB;
class EventServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        parent::boot();
        . . .
        Queue::looping(function ($event) {
            while (DB::transactionLevel() > 0) {
                DB::rollBack();
            }
        });
    }
    . . .
}
```

Остальные методы имеют тот же формат вызова, что и `looping()`:

- `before()` — привязывает обработчик к событию `JobProcessing`;
- `after()` — привязывает обработчик к событию `JobProcessed`;
- `exceptionOccured()` — привязывает обработчик к событию `JobExceptionOccured`;
- `failing()` — привязывает обработчик к событию `JobFailed`;
- `stopping()` — привязывает обработчик к событию `WorkerStopping`.



ГЛАВА 26

Cookie, сессии, всплывающие сообщения и криптография

26.1. Cookie

Cookie — произвольное значение, объемом не более 4096 байтов, сохраняемое на компьютере клиента под заданным уникальным именем. Cookie создается сервером после получения клиентского запроса и отправляется клиенту в составе серверного ответа. В cookie обычно хранятся служебные данные, настройки сайта и пр.

Каждое cookie при создании связывается с интернет-адресом создавшего его хоста и путем, по которому был отправлен создавший его запрос. В дальнейшем при отправке клиентом запроса на тот же хост и по тому же (или вложенному в него) пути все cookie, связанные с этими интернет-адресом и путем, отсылаются в составе запроса, чтобы сервер, работающий на хосте, смог прочитать хранящиеся в них значения. Также cookie имеет определенное время существования, по истечении которого оно удаляется веб-обозревателем.

26.1.1. Настройки cookie

Все cookie, создаваемые Laravel, шифруются для повышения безопасности. Однако если какие-либо cookie требуется обрабатывать в клиентских веб-сценариях, следует пометить эти cookie как не подлежащие шифрованию. Сделать это можно, занеся имена нужных cookie в массив, присвоенный защищенному свойству `except` посредника `App\Http\Middleware\EncryptCookies`. Пример:

```
class EncryptCookies extends Middleware {
    protected $except = ['counter', 'open_key'];
}
```

26.1.2. Создание cookie

Проще всего создать cookie и поместить его в отправляемый ответ, вызвав метод `cookie()` у объекта ответа при его создании:

```
cookie(<имя>, <значение>[, <время существования>=0[, <путь>=null[,
       <интернет-адрес хоста>=null[, <только по HTTPS?>=null[,
       <доступен только серверам?>=true[, <незашифрованный?>=false[,
       <отправлять другим хостам?>=null]]]]]]])
```

Имя cookie следует указывать в виде строки, а *значение* может быть любого типа. *Время существования* cookie задается в минутах — если указать 0, cookie будет удалено после закрытия веб-обозревателя.

Можно указать *путь* и *интернет-адрес хоста*, с которыми будет связано создаваемое cookie. Если они не заданы, cookie будет связано соответственно с текущими путем и интернет-адресом хоста.

По умолчанию, если клиентский запрос, создавший cookie, пришел серверу по протоколу HTTP, созданный cookie в дальнейшем будет отправляться серверу и по протоколу HTTP, и по протоколу HTTPS. Однако если клиентский запрос пришел по протоколу HTTPS, cookie будет отправляться серверу только по протоколу HTTPS. Дав параметру *только по HTTPS* значение `false`, можно указать в таком случае всегда отправлять cookie по любым протоколам.

По умолчанию создаваемое cookie будет доступно лишь серверам, но никак не клиентским веб-сценариям. Дав параметру *доступен только серверам* значение `false`, его можно сделать доступным и для веб-сценариев.

По умолчанию cookie, если его имя не указано в массиве из свойства `except` посредника `EncryptCookies`, шифруется для обеспечения безопасности. Дав параметру *незашифрованный* значение `true`, можно указать фреймворку не шифровать cookie (что пригодится, если это cookie должно быть доступно клиентским веб-сценариям).

Параметр *отправлять другим хостам* указывает, будет ли cookie доступно сайтам, располагающимся на других хостах. Он поддерживает следующие значения:

- `'strict'` — cookie не будет доступно другим сайтам;
- `'lax'` — cookie будет доступно другим сайтам лишь при переходе на них по гиперссылкам с текущего сайта;
- `'none'` — cookie будет доступно сторонним сайтам (небезопасно!);
- `null` — принятие решения, отправлять ли cookie сторонним сайтам, возлагается на веб-обозреватель (который, скорее всего, сделает cookie недоступным для «чужих» сайтов).

Пример создания cookie:

```
$counter = 0;
return response()->view('index', ['bbs' => $bbs])
                 ->cookie('counter', $counter, 60 * 24);
```

Также можно в произвольном месте кода создать объект cookie, а потом добавить его в ответ. Для создания объекта cookie применяется функция-хелпер `cookie()`, имеющая тот же формат вызова, что и одноименный метод, а для добавления *объекта cookie* в ответ следует вызвать у последнего метод `cookie(<объект cookie>)`. Пример:

```
$cookie = cookie('counter', $counter, 60 * 24);
. . .
return response()->view('index', ['bbs' => $bbs])->cookie($cookie);
```

Вместо метода `cookie()` в обоих случаях можно использовать полностью ему аналогичный метод `withCookie()`.

У метода `cookie()` есть недостаток — он поддерживается только классом `Response` (см. *разд. 9.5.1.2*). Так что добавить cookie в ответ, сгенерированный функцией `view()`, не получится.

В этом случае на помощь придет фасад `Illuminate\Support\Facades\Cookie` и поддерживаемые им методы. С их помощью можно создать cookie в любом месте кода и поместить их в особую очередь, откуда они будут перенесены в отправляемый ответ посредником `AddQueuedCookiesToResponse`. Вот эти методы:

- `queue()` — создает cookie и тут же помещает его в очередь для последующей отправки в составе серверного ответа. Формат вызова этого метода такой же, как и у метода `cookie()`. Пример:

```
use Illuminate\Support\Facades\Cookie;
. . .
Cookie::queue('counter', 0, 60 * 24);
```

- `queue(<`*объект cookie*`>)` — помещает заданное в виде *объекта* cookie в очередь для последующей отправки в составе серверного ответа. Альтернативный формат вызова этого метода. Пример:

```
$cookie = Cookie::make('counter', 0, 60 * 24);
Cookie::queue($cookie);
```

- `make()` — создает cookie и возвращает представляющий его объект. Формат вызова такой же, как и у метода `cookie()`.

Фасад `Cookie` поддерживает еще несколько полезных методов:

- `forever()` — создает и возвращает в виде объекта «вечное» cookie (с временем существования, равным 5 годам):

```
forever(<имя значения>, <значение>[, <путь>=null[,
        <интернет-адрес хоста>=null[, <только по HTTPS?>=null[,
        <доступен только серверам?>=true[, <незашифрованный?>=false[,
        <отправлять другим хостам?>=null]]]]]])
```

  В остальном он аналогичен методу `make()`. Пример:

```
Cookie::queue(Cookie::forever('counter', 0));
```

- `hasQueued(<`*имя*`>[, <`*путь*`>=null])` — возвращает *true*, если cookie с указанными *именем* и *путем* присутствует в очереди, и *false* — в противном случае;
- `unqueue(<`*имя*`>[, <`*путь*`>=null])` — удаляет из очереди cookie с заданными *именем* и *путем*.

26.1.3. Считывание cookie

Чтобы прочитать значение cookie с заданным *именем*, следует вызвать у объекта запроса метод `cookie()`:

```
cookie(<имя>[, <значение по умолчанию>=null])
```

Если cookie с указанным *именем* не существует, будет возвращено *значение по умолчанию*. Пример:

```
$counter = request()->cookie('counter', 0);
$counter++;
```

Вызвав метод `cookie()` без параметров, можно получить ассоциативный массив со всеми cookie, созданными сайтом. Ключи элементов этого массива соответствуют именам cookie, а значения элементов — суть значения этих cookie. Пример:

```
$cookies = request()->cookie();
$counter = $cookies['counter'];
```

Метод `hasCookie(<имя>)` запроса возвращает `true`, если cookie с заданным *именем* существует, и `false` — в противном случае.

Также можно использовать следующие методы фасада `Cookie`:

- `get()` — полностью аналогичен методу `cookie()` запроса:

  ```
  use Illuminate\Support\Facades\Cookie;
  . . .
  $counter = Cookie::get('counter', 0);
  ```

- `has()` — полностью аналогичен методу `hasCookie()` запроса.

26.1.4. Удаление cookie

Для удаления cookie с указанными *именем*, *путем* и *интернет-адресом* можно использовать метод `forget()` фасада `Cookie`:

```
forget(<имя>[, <путь>=null[, <интернет-адрес хоста>=null]])
```

Пример:

```
Cookie::forget('counter');
```

Фактически этот метод устанавливает время существования cookie в –5 лет, делая его тем самым устаревшим и подлежащим немедленному удалению веб-обозревателем.

Также можно использовать полностью аналогичный метод `expire()`.

Есть еще один способ удалить cookie — вызвать метод `withoutCookie()` у объекта ответа. Формат вызова этого метода такой же, как и у метода `forget()`. Пример:

```
return response()->view('index', ['bbs' => $bbs])
                 ->withoutCookie('counter');
```

26.2. Сессии

Сессия — это промежуток времени, в течение которого посетитель пребывает на текущем сайте. Сессия начинается, как только посетитель заходит на сайт, и завершается после его ухода.

Также *сессией* называются произвольные данные, относящиеся к посетителю, который в текущий момент находится на сайте, и сохраняемые на стороне сервера. Они записываются в виде пар *<имя>-<значение>*. Такие данные хранятся в течение определенного

времени, после чего удаляются, — это сделано для того, чтобы не занимать дисковое пространство устаревшими данными. Каждая сессия помечается уникальным идентификатором, сохраняемым в особом cookie (*cookie сессии*).

Поскольку данные сессии сохраняются на стороне сервера, в них можно хранить конфиденциальные сведения. В частности, подсистема разграничения доступа сохраняет в сессии сведения о текущем пользователе.

26.2.1. Подготовка к работе с сессиями

26.2.1.1. Настройки сессий

Настройки подсистемы серверных сессий хранятся в модуле config\session.php:

- `driver` — имя службы, в которой будут храниться сессии. Поддерживаются следующие службы:
  - `file` — обычные файлы в заданной папке;
  - `database` — таблица в реляционной базе данных;
  - `redis` — нереляционная база данных Redis;
  - `memcached` — популярный сервер кеширования Memcached;
  - `dynamodb` — нереляционная база данных Amazon DynamoDB;
  - `apc` — PHP-расширение Alternative PHP User Cache (APCu, **https://pecl.php.net/package/APCu**);
  - `cookie` — cookie сессии (т. е. в этом случае сессии фактически хранятся на стороне клиента);
  - `array` — массив в оперативной памяти.

  Значение берется из локальной настройки `SESSION_DRIVER`, имеющей значение `file`. По умолчанию: `file`;
- `lifetime` — время существования сессии в минутах после ухода посетителя с сайта. Значение берется из локальной настройки `SESSION_LIFETIME`, имеющей значение 120. По умолчанию: 120;
- `expire_on_close` — если `true`, сессия будет удалена сразу же после ухода посетителя с сайта. Если `false`, сессия будет существовать после ухода посетителя, пока не истечет время ее существования, указанное в настройке `lifetime`. По умолчанию: `false`;
- `encrypt` — если `true`, данные, сохраняемые в сессии, будут шифроваться, если `false` — не будут. Шифрование стоит включать, только если в сессиях планируется хранить какие-либо особо конфиденциальные сведения (например, номера документов, кредитных карт и т. п.), поскольку при его использовании снижается производительность. По умолчанию: `false`;
- `lottery` — вероятность, с которой при поступлении очередного запроса Laravel проведет удаление устаревших сессий, в виде «M шансов из N». Значение указывается в виде массива из двух элементов — значений M и N. По умолчанию: массив

`[2, 100]` (т. е. при поступлении очередного запроса есть 2 шанса из 100, что фреймворк удалит устаревшие сессии).

Следующая настройка используется только службой `file`:

- `files` — полный путь к папке, в которой будут храниться файлы с сессиями (по умолчанию: полный путь к папке storage\framework\sessions).

Следующая настройка используется только службами `database` и `redis`:

- `connection` — имя базы данных из указанных в настройках `database.connections` (см. *разд. 3.4.2.4*) или `database.redis` (см. *разд. 25.1.2*). Значение берется из локальной настройки `SESSION_CONNECTION`, изначально отсутствующей в файле .env. По умолчанию: `null` (база данных по умолчанию).

Следующая настройка используется только службой `database`:

- `table` — имя таблицы, в которой будут храниться сессии (по умолчанию: `sessions`).

Следующая настройка используется службами `memcached`, `dynamodb` и `apc`:

- `store` — имя службы кеша, используемой для хранения сессий, из приведенных в настройке `cache.stores`. Если указано значение `null`, будет использована служба с именем, совпадающим с именем выбранной службы сессий. Значение берется из локальной настройки `SESSION_STORE`, изначально отсутствующей в файле .env. По умолчанию: `null`.

Следующие настройки задают параметры cookie сессии:

- `cookie` — имя cookie сессии. По умолчанию: строка, составленная из названия проекта (берется из локальной настройки `APP_NAME`), слов `laravel` и `session`, записанная в стиле snake_case;
- `path` — путь, с которым будут связаны cookie (по умолчанию: /, т. е. «корень» сайта);
- `domain` — интернет-адрес хоста, с которым будут связаны cookie. Значение берется из локальной настройки `SESSION_DOMAIN`, изначально отсутствующей в файле .env. По умолчанию: `null` (обозначает текущий интернет-адрес);
- `secure` — аналогична параметру *только по HTTPS* метода `cookie()` (см. *разд. 26.1.1*). Значение берется из локальной настройки `SESSION_SECURE_COOKIE`, изначально отсутствующей в файле .env. По умолчанию: `null`;
- `http_only` — аналогична параметру *доступен только серверам* метода `cookie()` (по умолчанию: `true`);
- `same_site` — аналогична параметру *отправлять другим хостам* метода `cookie()` (по умолчанию: `'lax'`).

26.2.1.2. Создание таблицы для хранения сессий

Если для хранений сессий выбрана служба `database`, т. е. обычная реляционная база данных, в этой базе нужно создать таблицу, в которой и будут храниться сессии.

Чтобы сгенерировать миграцию, создающую эту таблицу, достаточно набрать команду:

```
php artisan session:table
```

Созданная миграция получит имя формата *<временна́я отметка>*_create_sessions_table.php. К сожалению, она в любом случае создает таблицу с именем `sessions`. Так что, если в настройке `session.table` было указано другое имя для таблицы сессий, код миграции придется исправить вручную.

После создания миграции следует выполнить миграции.

26.2.2. Работа с сессиями

26.2.2.1. Запись данных в сессию и их изменение

Проще всего записать какие-либо значения в сессию, вызвав функцию-хелпер `session(<массив значений>)`. В заданном ассоциативном *массиве значений* ключи элементов определят имена записываемых в сессию значений, а значения элементов — сами записываемые значения. Пример:

```
session(['bb.id' => $bb->id, 'bb.title' => $bb->title, 'bb.price' => 0]);
```

Несколько более высокое быстродействие обеспечивают методы объекта сессии. Этот объект можно получить, вызвав у объекта клиентского запроса метод `session()`. Сами методы сессии, записывающие данные, приведены далее:

- `put()` — записывает в сессию заданное значение или значения, в зависимости от формата вызова:

  ```
  put(<имя значения>, <значение>)
  put(<ассоциативный массив значений>)
  ```

  Первый формат сохраняет в сессии одно *значение* под заданным *именем*:

  ```
  $store = request()->session();
  $store->put('bb.id', $bb->id);
  ```

  Второй формат позволяет сохранить произвольное количество значений и аналогичен таковому у функции `session()`:

  ```
  $store->put(['bb.title' => $bb->title, 'bb.price' => 0]);
  ```

- `replace(<ассоциативный массив значений>)` — аналогичен второму формату вызова метода `put()`.

Следующие методы службы сессий позволяют изменить ранее записанные данные:

- `push(<имя значения>, <новый элемент>)` — добавляет в массив, хранящийся в сессии под заданным *именем*, указанный *новый элемент*:

  ```
  $store->put('platforms', ['PHP', 'Laravel', 'MySQL']);
  $store->push('platforms', 'Redis');
  ```

- `increment(<имя>[, <величина>=1])` — увеличивает значение, хранящееся в сессии под заданным *именем*, на указанную *величину*:

  ```
  $store->increment('bb.price');
  $store->increment('bb.price', 9);
  ```

- `decrement(<имя>[, <величина>=1])` — уменьшает значение, хранящееся в сессии под заданным *именем*, на указанную *величину*:

  ```
  $store->decrement('bb.price', 10);
  ```

26.2.2.2. Чтение данных из сессии

Проще всего прочитать из сессии значение с заданным *именем*, вызвав функцию-хелпер `session()`:

```
session(<имя>[, <значение по умолчанию>=null])
```

Если значение с заданным *именем* в сессии отсутствует, будет возвращено *значение по умолчанию*. Пример:

```
$title = session('bb.title');
```

Как и при записи данных, несколько большее быстродействие обеспечивают следующие методы объекта сессии:

- `get()` — возвращает значение с заданным именем или значение по умолчанию, если такого имени в сессии нет. Формат вызова такой же, как и у функции `session()`. Пример:

  ```
  $store = request()->session();
  $title = $store->get('bb.title');
  ```

  Вторым параметром можно указать анонимную функцию, не принимающую параметров и возвращающую результат, который и станет значением по умолчанию;

- `exists(<имя>)` — возвращает `true`, если в сессии существует значение с заданным *именем*, и `false` — в противном случае:

  ```
  if ($store->exists('bb.price'))
      $result = 'Цена в сессии сохранена';
  ```

- `has(<имя>)` — возвращает `true`, если в сессии существует значение с заданным *именем*, не равное `null`, и `false` — в противном случае;

- `missing(<имя>)` — возвращает `true`, если значение с заданным *именем* равно `null` или вообще отсутствует в сессии, и `false` — в противном случае;

- `all()` — возвращает все значения, сохраненные в сессии, в виде ассоциативного массива:

  ```
  $sessionData = $store->all();
  $title = $sessionData['bb.title'];
  ```

- `only(<массив имен>)` — возвращает только значения с именами, содержащимися в заданном *массиве*, в виде ассоциативного массива:

  ```
  $sessionData = $store->only(['bb.title', 'bb.price']);
  ```

26.2.2.3. Удаление данных из сессии

Для удаления данных из сессии применяются следующие методы, поддерживаемые объектом сессии:

- `pull(<имя>[, <значение по умолчанию>=null])` — возвращает значение с заданным *именем*, после чего удаляет его из сессии. Если значения с таким *именем* нет, возвращает *значение по умолчанию*. Пример:

  ```
  $store = request()->session();
  $price = $store->pull('bb.price');
  ```

- `remove(<имя>)` — то же самое, что и `pull()`, только в случае отсутствия в сессии значения с заданным именем всегда возвращает `null`;
- `forget(<имя>|<массив имен>)` — удаляет из сессии значение с заданным *именем* или значения с именами, приведенными в заданном *массиве*:

```
$store->forget(['bb.title', 'bb.price']);
```

- `flush()` — удаляет все значения из сессии.

26.2.2.4. Блокировка сессий

Может случиться так, что одно и то же значение, хранящееся в сессии, попытаются изменить сразу два процесса (например, два разных действия контроллера или два экземпляра одного и того же действия), обрабатывающие два разные запроса с одного и того же сайта. Попытка одновременно изменить значение из двух процессов может привести к его потере.

Чтобы предотвратить подобные коллизии, фреймворк позволяет временно блокировать сессии, позволяя изменять одно значение только одному процессу. Остальные процессы, желающие изменить то же значение, будут вынуждены ждать, пока первый процесс не закончит работу или пока не истечет время действия блокировки.

> ***Инструменты для блокировки сессий…***
>
> …используют для работы распределенные блокировки (см. *разд. 29.1.3*).

Блокировка сессии реализуется методом `block()`, вызываемым у объекта маршрута (о маршрутизации рассказывалось в *главе 8*):

```
block([<время блокировки>=10[, <время ожидания снятия блокировки>=10]])
```

Оба значения *времени* задаются в секундах.

Сессия будет заблокирована, пока не завершится работающий с ней поток или пока не истечет заданное *время блокировки*.

Если поток не сможет заблокировать сессию (поскольку она уже заблокирована другим потоком), он ждет в течение указанного *времени ожидания*. Если за это время блокировка не будет снята, будет возбуждено исключение `Illuminate\Contracts\Cache\LockTimeoutException`.

Метод `block()` вызывается у маршрута, ведущего на действие, в котором требуется заблокировать сессию. Примеры:

```
// Параметры по умолчанию: блокируем на 10 секунд, ждем 10 секунд
Route::post('/profile', ... )->block();
// Блокируем на 2 секунды, ждем 10 секунд
Route::post('/profile', ... )->block(2);
// Блокируем на 2 секунды, ждем 1 секунду
Route::post('/profile', ... )->block(2, 1);
```

26.2.2.5. Предотвращение сетевых атак на сессии

Для предотвращения некоторых сетевых атак рекомендуется после успешного входа на сайт перегенерировать идентификатор сессии, а после выхода с сайта — очистить сес-

сию и перегенерировать электронный жетон безопасности. Если для выполнения входа и выхода используется контроллер `App\Http\Controllers\Auth\LoginController`, создаваемый в составе проекта командой `ui:auth` утилиты artisan (см. *разд. 13.2*), явно выполнять эти действия не нужно — это сделает упомянутый ранее контроллер. Если же вход и выход реализуются низкоуровневыми инструментами (описанными в *разд. 19.1*), придется выполнять эти действия явно, вызывая у объекта сессии следующие методы:

- `regenerate()` — повторно генерирует идентификатор сессии, сохраняя записанные в ней данные. Вызывается после успешного входа. Пример:

```
public function login(Request $request) {
    $email = $request->email;
    $password = $request->password;
    if (Auth::attempt(['email' => $email, 'password' => $password])) {
        $request->session()->regenerate();
        return redirect()->intended('/');
    }
}
```

- `invalidate()` — очищает сессию и повторно генерирует ее идентификатор. Вызывается после успешного выхода;
- `regenerateToken()` — перегенерирует и записывает в сессию электронный жетон безопасности, используемый для защиты от межсайтовых атак. Вызывается после успешного выхода и вызова метода `invalidate()`.

  Пример:

```
public function logout(Request $request) {
    Auth::logout();
    $request->session()->invalidate();
    $request->session()->regenerateToken();
    return redirect('/');
}
```

26.3. Всплывающие сообщения

Всплывающие сообщения записываются в сессию в процессе обработки текущего клиентского запроса и остаются доступными в течение следующего клиентского запроса, после чего автоматически удаляются. Они применяются в случаях, когда действие контроллера выполняет некую операцию (например, добавление объявления), после чего производит перенаправление на какую-либо веб-страницу, и на этой странице требуется вывести сообщение об успешном выполнении операции.

Чтобы записать в сессию новое всплывающее сообщение под заданным *именем*, следует вызвать у объекта сессии метод `flash(<имя>, <текст сообщения>)`:

```
$session = request()->session();
$session->flash('success', 'Объявление успешно добавлено');
return redirect('/');
```

Создать всплывающее сообщение также можно, вызвав у объекта перенаправления метод `with()`. Он имеет тот же формат вызова, что и метод `flash()`. Пример:

```
return redirect('/')->with('success', 'Объявление успешно добавлено');
```

Извлечь такое сообщение в коде шаблона можно, воспользовавшись функцией-хелпером `session()`, описанной в *разд. 26.2.2.2*, и указав в качестве единственного параметра имя нужного сообщения:

```
@if (session('success'))
<div class="alert alert-success">
    {{ session('success') }}
</div>
@endif
```

Объект сессии также поддерживает следующие методы, которые могут оказаться полезными в ряде случаев:

- `reflash()` — продлевает срок хранения всех всплывающих сообщений, имеющихся в сессии, делая их доступными на протяжении всех последующих клиентских запросов.

  Для удаления всплывающих сообщений с продленным сроком хранения следует использовать методы `forget()` или `flush()`, описанные в *разд. 26.2.2.3*;

- `keep(<`*массив имен*`>)` — продлевает срок хранения только всплывающих сообщений с именами, приведенными в заданном *массиве*:

  ```
  $session->keep(['error', 'warning']);
  ```

- `now()` — записывает в сессию всплывающее сообщение и делает его доступным только на протяжении текущего клиентского запроса. Формат вызова такой же, как и у метода `flash()`.

26.4. Криптография

26.4.1. Шифрование данных

Шифрование заданного *значения* выполняется вызовом у фасада `Illuminate\Support\Facades\Crypt` метода `encrypt(<`*значение*`>)`:

```
>>> use Illuminate\Support\Facades\Crypt;
>>> $cr1 = Crypt::encrypt(123);
=> "eyJpdiI6Ikw5aUhwN2FhRlpQTzk5ZU1ZY2 ... OGEyZjcxMGQxMjAwYTlhYSJ9"
```

Шифруемое *значение* может быть любого типа, поскольку перед шифрованием оно сериализуется в строковое представление:

```
>>> $cr2 = Crypt::encrypt([1, 2, 3, 4]);
=> "eyJpdiI6IjhER0ExQTNwRDdsTzdNWTRzSmJ3a ... OTJiNDExYjRiYmMzOTUifQ=="
```

Дешифрование *зашифрованного значения* производится вызовом метода `decrypt(<`*зашифрованное значение*`>)` того же фасада:

```
>>> Crypt::decrypt($cr1);
=> 123
>>> Crypt::decrypt($cr2);
=> [ 1, 2, 3, 4 ]
```

Также можно использовать методы `encryptString()` и `decryptString()` соответственно. Они аналогичны методам `encrypt()` и `decrypt()`, только позволяют шифровать лишь значения элементарных типов (массивы и объекты шифровать с их помощью нельзя).

Наконец, можно использовать функции-хелперы `encrypt()` и `decrypt()` соответственно. Их вызвать проще, чем методы фасада, однако они немного медленнее. Пример:

```
$cr1 = encrypt(123);
. . .
if (decrypt($cr1) == 123)
    . . .
```

Шифрование выполняется на основе секретного ключа из рабочей настройки `app.key` с применением алгоритма, заданного в настройке `app.cipher` (см. *разд. 3.4.2.3*).

26.4.2. Хеширование и сверка паролей

26.4.2.1. Настройки хеширования

Настройки подсистемы хеширования паролей хранятся в модуле config\hashing.php:

- `driver` — обозначение алгоритма, используемого для хеширования: `'bcrypt'` (Bcrypt), `'argon'` (Argon2i) и `'argon2id'` (Argon2id). По умолчанию: `'bcrypt'`;
- `bcrypt` — настройка алгоритма Bcrypt:
  - `rounds` — алгоритмическая сложность в виде целого числа. Значение берется из локальной настройки `BCRYPT_ROUNDS`, изначально отсутствующей в файле .env. По умолчанию: 10;
- `argon` — настройки алгоритмов Argon2i и Argon2id:
  - `memory` — максимальный объем памяти, отводимый под расчет хеша, в виде целого числа в Кбайт (по умолчанию: 65 536);
  - `threads` — максимальное количество потоков, отводимых под вычисление хеша, в виде целого числа (по умолчанию: 1);
  - `time` — максимальное время вычисления хеша в виде целого числа в секундах (по умолчанию: 4).

26.4.2.2. Хеширование и сверка

Для хеширования заданного *пароля* достаточно вызвать у фасада `Illuminate\Support\Facades\Hash` метод `make()`:

```
hash(<хешируемый пароль>[, <параметры алгоритма>=[]])
```

Хешированный пароль возвращается в качестве результата. Пример:

```
>>> use Illuminate\Support\Facades\Hash;
>>> $hp1 = Hash::make('mypassword');
=> "$2y$10$bPNER31GCCl9sEpfKOytk.u.eEf6Erd08.jLdmNVwEI1qtM1WUiZq"
```

Можно указать *параметры алгоритма* в виде ассоциативного массива. Ключами элементов этого массива должны быть имена параметров, указываемые в настройках

hashing.bcrypt и hashing.argon (см. *разд. 26.4.2.1*), а значения элементов зададут значения параметров. Пример:

```
>>> use Illuminate\Support\Facades\Hash;
>>> $hp2 = Hash::make('mypassword', ['rounds' => 8]);
=> "$2y$08$r/mwe23KJcs0IfwnLHremumiz4r8n/Gk4Lm1ziA5zKU/VW8ziHYrq"
```

Для хеширования заданного *пароля* с указанными *параметрами* с применением алгоритма Bcrypt можно использовать функцию-хелпер bcrypt() — это позволит несколько сократить код:

```
bcrypt(<хешируемый пароль>[, <параметры алгоритма>=[]])
```

Сверка *хешированного пароля* с *нехешированным* производится методом check() фасада Hash:

```
check(<нехешированный пароль>, <хешированный пароль>[,
                               <параметры алгоритма>=[]])
```

Параметры алгоритма указываются в том же формате, что и у метода make(). Примеры:

```
>>> Hash::check('mypassword', $hp1);
=> true
>>> Hash::check('hispassword', $hp2, ['rounds' => 8]);
=> false
```

Метод needsRehash() того же фасада возвращает true, если заданный *хешированный пароль* требуется перехешировать, и false — в противном случае:

```
needsRehash(<хешированный пароль>[, <параметры алгоритма>=[]])
```

26.4.3. Генерирование подписанных интернет-адресов

Подписанный интернет-адрес включает цифровую подпись, заданную в GET-параметре signature. Благодаря ей фреймворк сможет удостовериться, что интернет-адрес, по которому на сайт пришел посетитель, не был подделан. Подписанные интернет-адреса используются для создания дополнительной защиты от сетевых атак, связанных с подделкой интернет-адресов.

Сгенерировать подписанный интернет-адрес, ведущий на маршрут с указанным *именем*, можно, вызвав один из следующих методов фасада Illuminate\Support\Facades\URL:

- signedRoute() — генерирует и возвращает подписанный интернет-адрес:

  ```
  signedRoute(<имя маршрута>[,
              <ассоциативный массив со значениями URL-параметров>=[][,
              <время актуальности>=null[,
              <полный интернет-адрес?>=true]]])
  ```

  Пример:

  ```
  use Illuminate\Support\Facades\URL;
  . . .
  $url = URL::signedRoute('unsubscribe', ['user' => $user->id]);
  ```

  Если указано *время актуальности*, будет сгенерирован *временный интернет-адрес*, действительный до этого *времени*. *Время актуальности* можно указать в любом фор-

мате, поддерживаемом PHP, или в виде объекта класса `Carbon`. Время актуальности записывается в GET-параметре `expires`, добавляемом в интернет-адрес. Пример:

```
$url = URL::signedRoute('unsubscribe', ['user' => $user->id],
                                       now()->addDays(2));
```

По умолчанию будет сгенерирован полный интернет-адрес. Для генерирования сокращенного интернет-адреса следует дать параметру *полный интернет-адрес* значение `false`;

- `temporarySignedRoute()` — то же самое, что и `signedRoute()`, только имеет другой формат вызова, более подходящий для генерирования временных интернет-адресов:

```
temporarySignedRoute(<имя маршрута>, <время актуальности>,
                     <ассоциативный массив с URL-параметрами>=[][,
                     <полный интернет-адрес?>=true]])
```

Для проверки, не был ли подделан интернет-адрес, по которому посетитель пришел на сайт, проще всего использовать посредник `Illuminate\Routing\Middleware\ValidateSignature`, имеющий обозначение `signed`. Его достаточно связать с нужным маршрутом. Пример:

```
Route::get('/unsubscribe/{user}',
           [AccountController::class, 'unsubscribe'])
    ->name('unsubscribe')->middleware('signed');
```

Если интернет-адрес был подделан или его время актуальности вышло, посредник возбудит исключение `Illuminate\Routing\Exceptions\InvalidSignatureException`, что приведет к выдаче сообщения об ошибке с кодом 403 (доступ запрещен).

Для проверки корректности подписанного интернет-адреса также можно использовать следующие методы:

- фасада `URL`:
  - `hasCorrectSignature(<запрос>)` — возвращает `true`, если цифровая подпись, находящаяся в составе извлеченного из заданного *запроса* интернет-адреса, соответствует этому интернет-адресу, и `false` — в противном случае. Формат вызова:

```
hasCorrectSignature(<запрос>[, <полный интернет-адрес?>=true[,
          <массив с именами исключаемых GET-параметров>=[]]])
```

Запрос задается в виде объекта.

Изначально при проверке корректности фреймворк принимает во внимание весь интернет-адрес, включая обозначение протокола, имя хоста, номер TCP-порта и путь. Если нужно, чтобы он принимал во внимание только путь, следует дать параметру *полный интернет-адрес* значение `false`.

Может случиться так, что в уже сгенерированный подписанный интернет-адрес будут добавлены те или иные GET-параметры (что может сделать какая-либо клиентская JavaScript-библиотека). В таком случае необходимо указать Laravel, чтобы он при проверке корректности интернет-адреса не принимал во внимание эти GET-параметры. Для этого достаточно занести имена исключаемых из проверки GET-параметров в передаваемый методу *массив*.

Пример:

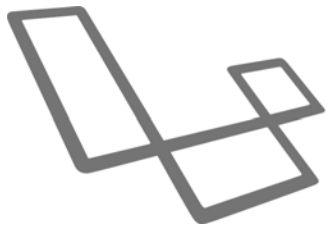
```
use Illuminate\Support\Facades\URL;
. . .
if (URL::hasCorrectSignature(request()))
    // Цифровая подпись соответствует интернет-адресу
```

- `signatureHasNotExpired(<`*запрос*`>)` — возвращает `true`, если время актуальности, записанное в извлеченном из заданного *запроса* временном интернет-адресе, еще не истекло, и `false` — в противном случае. *Запрос* указывается в виде объекта;
- `hasValidSignature(<`*запрос*`>)` — объединяет методы `hasCorrectSignature()` и `signatureHasNotExpired()`. Формат вызова такой же, как и у метода `hasCorrectSignature()`;

❒ объекта запроса:

- `hasValidSignature([<`*полный интернет-адрес?*`>=true])` — аналогичен одноименному методу фасада `URL`:

```
if (request()->hasValidSignature())
    // Цифровая подпись корректна, и интернет-адрес не устарел
```



ГЛАВА 27

Планировщик заданий

Часто бывает необходимо регулярно и в заданные моменты времени выполнять какие-либо действия (например, очищать базу данных от устаревших объявлений или «мертвых душ» — пользователей, давно зарегистрировавшихся, но ни разу не заходивших на сайт). Laravel предлагает свой собственный *планировщик заданий*, выполняющий по расписанию действия подобного рода, которые оформляются в виде *заданий планировщика*.

27.1. Создание заданий планировщика

27.1.1. Как пишутся задания планировщика?

Задания планировщика создаются в «корневом» классе утилиты artisan `App\Console\Kernel` в защищенном методе `schedule()`. Этот метод в качестве параметра принимает объект класса `Illuminate\Console\Scheduling\Schedule`, представляющий сам планировщик заданий.

Задания создаются с помощью следующих методов планировщика:

- `call(<`*анонимная функция*`>)` — создает задание, реализованное указанной *анонимной функцией*, которая не должна принимать параметры и возвращать результат. В качестве результата метод возвращает объект задания, у которого можно вызвать методы, определяющие параметры задания, — в частности, расписание его запуска (эти методы будут рассмотрены позже).

  Пример задания планировщика, удаляющего объявления, которые помечены как не подлежащие публикации, и выполняющегося ежемесячно (указывается вызовом метода `monthly()` у объекта задания, возвращенного методом `call()`):

```
use App\Models\Bb;
class Kernel extends ConsoleKernel {
    . . .
    protected function schedule(Schedule $schedule) {
        $schedule->call(function () {
            Bb::where('publish', false)->delete();
```

```
        })->monthly();
    }
    . . .
}
```

Вместо *анонимной функции* можно указать объект какого-либо класса, содержащего метод `__invoke()`, в котором записан код, реализующий задание:

```
$schedule->call(new DeleteUnpublishedBbs)->monthly();
```

- `command()` — создает задание, реализуемое указанной *командой* утилиты artisan. Поддерживает два формата вызова:

```
command(<команда>)
command(<путь к классу команды>[, <массив командных ключей>=[]])
```

В первом формате указывается сама запускаемая *команда* вместе с ключами, заданная в виде строки. Пример задания, реализованного командой `queue:work` с ключами `--queue=mail` и `--stop-when-empty` и выполняющегося каждые 5 минут (задается вызовом метода `everyFiveMinutes()`):

```
protected function schedule(Schedule $schedule) {
    $schedule->command('queue:work --queue=mail --stop-when-empty')
             ->everyFiveMinutes();
}
```

Во втором формате отдельно указываются *путь к классу*, реализующему нужную команду artisan, и *массив командных ключей*:

```
use Illuminate\Queue\Console\WorkCommand;
. . .
protected function schedule(Schedule $schedule) {
    $schedule->command(WorkCommand::class,
                       ['--queue=mail', '--stop-when-empty'])
             ->everyFiveMinutes();
}
```

- `exec()` — создает задание планировщика, выполняющее команду операционной системы. Форматы вызова:

```
exec(<команда>)
exec(<путь к исполняемому файлу команды>[, <массив командных ключей>=[]])
```

Можно указать либо саму *команду* вместе с ключами в виде строки, либо отдельно *путь к исполняемому файлу команды* и *массив с ключами*. Примеры:

```
$schedule->exec('c:/scripts/clearfiles.cmd *.tmp *.bak')->monthly();
$schedule->exec('c:/scripts/clearfiles.cmd', ['*.tmp', '*.bak'])
         ->monthly();
```

- `job()` — создает задание планировщика на основе *отложенного задания*, указанного в виде объекта:

```
job(<отложенное задание>[, <имя очереди>=null[,
                           <имя службы очередей>=null]])
```

Пример задания планировщика, создаваемого на основе отложенного задания `OldUsersDeleteJob` и выполняемого ежегодно (задается вызовом метода `yearly()`):

```
use App\Jobs\OldUsersDeleteJob;
. . .
$schedule->job(new OldUsersDeleteJob)->yearly();
```

Изначально отложенное задание помещается в очередь по умолчанию службы очередей по умолчанию. Но можно указать *имена* других *очереди* и *службы очередей*. Пример:

```
$schedule->job(new OldUsersDeleteJob, 'old-data', 'database')
         ->yearly();
```

27.1.2. Параметры заданий планировщика

Методы планировщика, описанные в *разд. 27.1.1*, в качестве результата возвращают объект задания планировщика. Вызывая методы этого объекта, можно указать параметра задания — в частности, расписание его запуска.

27.1.2.1. Расписание запуска заданий

Задать расписание запуска задания можно с помощью следующих методов:

- `everyMinute()` — задание будет выполняться ежеминутно:
  ```
  $schedule->call( ... )->everyMinute();
  ```
- `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()` — ежечасно, в начале каждого часа;
- `hourlyAt(<количество минут>)` — ежечасно, спустя указанное *количество минут* после начала часа:
  ```
  $schedule->call( ... )->hourlyAt(15);
  ```
- `everyTwoHours()` — каждые 2 часа;
- `everyThreeHours()` — каждые 3 часа;
- `everyFourHours()` — каждые 4 часа;
- `everySixHours()` — каждые 6 часов;
- `daily()` — ежедневно, в полночь;
- `dailyAt(<время>)` — ежедневно, в указанное *время*. *Время* задается в виде строки в формате `<часы>:<минуты>[:<секунды>]`. Пример:
  ```
  $schedule->call( ... )->dailyAt('11:40');
  ```
- `at()` — то же самое, что и `dailyAt()`;

- `twiceDaily([<время 1>=1[, <время 2>=13]])` — дважды в сутки, в моменты *времени 1* и *2*, указываемые в виде количества часов, прошедших с начала суток:

  ```
  $schedule->call( ... )->twiceDaily(0, 12);
  ```

- `weekly()` — еженедельно, в полночь воскресенья;
- `weeklyOn(<номер дня недели>[, <время>='0:0'])` — еженедельно, в день с указанным *номером* и в заданное *время*. Воскресенье обозначается номером 0, понедельник — 1, вторник — 2 и т. д. Пример запуска задания в субботу, в 2:00:

  ```
  $schedule->call( ... )->weeklyOn(6, '2:00');
  ```

  Вместо номеров дней недели можно использовать следующие константы класса `Schedule`: `MONDAY` (понедельник), `TUESDAY` (вторник), `WEDNESDAY` (среда), `THURSDAY` (четверг), `FRIDAY` (пятница), `SATURDAY` (суббота) и `SUNDAY` (воскресенье):

  ```
  use Illuminate\Console\Scheduling\Schedule;
  . . .
  $schedule->call( ... )->weeklyOn(Schedule::SATURDAY, '2:00');
  ```

- `monthly()` — ежемесячно, в полночь первого дня месяца;
- `monthlyOn([<число>=1[, <время>='0:0']])` — ежемесячно, в заданное *число* и *время*. Пример запуска задачи в 22:00 15-го числа:

  ```
  $schedule->call( ... )->monthlyOn(15, '22:00');
  ```

- `lastDayOfMonth([<время>='0:0'])` — в последний день каждого месяца, в заданное *время*. Пример запуска задачи в 23:00:

  ```
  $schedule->call( ... )->lastDayOfMonth('23:00');
  ```

- `twiceMonthly([<число 1>=1[, <число 2>=16]])` — дважды в месяц, в дни, заданные *числами 1* и *2*;
- `quarterly()` — ежеквартально, в полночь первого дня квартала;
- `yearly()` — ежегодно, в полночь 1 января;
- `yearlyOn()` — ежегодно, в указанное *число* месяца с заданным *номером*, в указанное *время*:

  ```
  yearlyOn([<номер месяца>=1[, <число>=1[, <время>='0:0']]])
  ```

  Пример:

  ```
  $schedule->call( ... )->yearlyOn(5, 31, '7:30');
  ```

В качестве результата эти методы возвращают текущий объект задания. Вызывая у этого объекта представленные далее методы, можно задать дополнительные параметры расписания:

- `weekdays()` — указывает выполнять задание только в будние дни:

  ```
  $schedule->call( ... )->weekly()->weekdays();
  ```

- `weekends()` — только в выходные;
- `mondays()` — по понедельникам:

  ```
  $schedule->call( ... )->weekly()->mondays();
  ```

- `tuesdays()` — по вторникам:

```
$schedule->call( ... )->hourlyAt(30)->tuesdays();
```

- `wednesdays()` — по средам;
- `thursdays()` — по четвергам;
- `fridays()` — по пятницам;
- `saturdays()` — по субботам;
- `sundays()` — по воскресеньям;
- `days()` — только в дни с заданными *номерами* (воскресенье — 0, понедельник — 1, вторник — 2 и т. д.). Поддерживаются два формата вызова:

```
days(<номер дня 1>, <номер дня 2>, ... <номер дня n>)
days(<массив с номерами дней>)
```

  Вместо номеров дней можно указывать константы класса `Schedule`, описанные ранее. Примеры:

```
$schedule->call( ... )->hourly()->days(1, 3, 5);
$schedule->call( ... )->hourly()
         ->days([Schedule::MONDAY, Schedule::WEDNESDAY, Schedule::FRIDAY]);
```

- `between(<время 1>, <время 2>)` — только в промежутке от *времени 1* до *времени 2*. *Время* задается в том же формате, что и в вызове метода `dailyAt()`. Пример:

```
$schedule->call( ... )->hourly()->between('8:30', '17:00');
```

- `unlessBetween()` — наоборот, в любое время, кроме промежутка между временем 1 и временем 2. Формат вызова такой же, как и у метода `between()`;
- `timezone(<обозначение временно́й зоны>)` — задает временну́ю зону в виде строкового *обозначения*. Указанная временная зона будет использоваться вместо заданной в рабочей настройке `app.timezone`. Пример:

```
$schedule->call( ... )->dailyAt('11:40')
                      ->timezone('Europe/Volgograd');
```

  Если все задания планировщика должны использовать временну́ю зону, отличную от указанной настройках проекта, можно объявить в классе `App/Console/Kernel` защищенный метод `scheduleTimezone()`, который в качестве результата должен возвращать обозначение нужной временно́й зоны:

```
class Kernel extends ConsoleKernel {
    . . .
    protected function scheduleTimezone() {
        return 'Europe/Volgograd';
    }
}
```

27.1.2.2. Дополнительные параметры заданий

У заданий можно задать дополнительные параметры, вызывая следующие методы:

- `name(<имя>)` — задает *имя* для задания. *Имя* может быть произвольным и служит исключительно для удобства разработчиков и администраторов сайта;

- `description()` — то же самое, что и `name()`;
- `when(<анонимная функция>)` — выполнять задание только в том случае, если указанная *анонимная функция* вернет значение `true`. *Анонимная функция* не должна принимать параметры. Пример:

```
$schedule->call( ... )->monthly()->when(function () {
    return (Bb::where('publish', false)->count() > 0);
});
```

- `skip()` — выполнять задание только в том случае, если заданная анонимная функция вернет значение `false`. Формат вызова такой же, как и у метода `when()`;
- `environments()` — выполнять задание, только когда сайт работает в указанных *режимах*. Поддерживаются два формата вызова:

```
environments(<режим 1>, <режим 2>, ... <режим n>)
environments(<массив режимов>)
```

Примеры:

```
$schedule->call( ... )->hourly()->environments('local', 'testing');
$schedule->call( ... )->hourly()
                      ->environments(['local', 'testing']);
```

По умолчанию экземпляр очередного задания планировщика будет запускаться на выполнение даже в том случае, если все еще выполняется ее ранее запущенный экземпляр. Таким образом, может возникнуть ситуация, когда несколько экземпляров одного и того же задания работают одновременно (накладываются друг на друга). В ряде случаев это нежелательно;

- `withoutOverlapping([<промежуток времени>=1440])` — указывает по возможности избегать наложения отдельных экземпляров одного и того же задания друг на друга. Если же в момент запуска очередного экземпляра задания какой-либо ранее запущенный его экземпляр еще выполняется, Laravel выждет указанный *промежуток времени* (в минутах) перед его запуском. Пример:

```
$schedule->call( ... )->hourly()->withoutOverlapping(10);
```

> ***Для предотвращения наложения заданий…***
> …планировщик использует распределенные блокировки (см. *разд. 29.1.3*).

По умолчанию команды утилиты artisan и операционной системы запускаются планировщиком последовательно друг за другом;

- `runInBackground()` — предписывает выполнять команду утилиты artisan или операционной системы в фоновом режиме, благодаря чему несколько команд могут быть выполнены одновременно.

Если несколько копий сайта работают на разных компьютерах и запросы на них распределяются каким-либо балансировщиком нагрузки, по умолчанию задания планировщика запускаются на всех этих компьютерах. В ряде случаев это может быть излишне;

- `onOneServer()` — указывает в таких случаях выполнять задание только на одном компьютере — том, который первым запустил задание на выполнение. Чтобы этот метод сработал, необходимо выполнить два условия:

- задание должно иметь уникальное имя. Задать его можно вызовом метода `name()` или `description()`. Пример:

```
$schedule->call( ... )->dailyAt('17:00')->fridays()
        ->name('Формирование недельного отчета')->onOneServer();
```

- все копии сайта должны использовать одну и ту же службу кеширования по умолчанию (о кешировании будет рассказано в *главе 29*).

ДЛЯ ПРЕДОТВРАЩЕНИЯ ЗАПУСКА ЗАДАНИЙ НА РАЗНЫХ СЕРВЕРАХ…
…планировщик использует распределенные блокировки (см. *разд. 29.1.3*).

По умолчанию задания планировщика не выполняются, если сайт находится в режиме обслуживания (см. *главу 35*);

- ❐ `evenInMaintetanceMode()` — предписывает выполнять задание, даже если сайт находится в режиме обслуживания.

27.1.3. Обработка вывода, генерируемого заданиями планировщика

Если команда утилиты artisan или операционной системы, запущенная в качестве задания планировщика, генерирует какой-либо вывод (например, сообщение об успешном или неуспешном выполнении), его можно записать в указанный файл или отправить по электронной почте. Для этого применяются следующие методы, поддерживаемые объектом задания:

- ❐ `sendOutputTo()` — записывает вывод, сгенерированный заданием, в файл с указанным *путем*. Формат вызова:

```
sendOutputTo(<полный путь к файлу>[, <добавлять в файл?>=false])
```

Пример:

```
$schedule->call( ... )->hourlyAt(15)
        ->sendOutputTo(storage_path('logs/scheduler.log'));
```

Если файл с заданным *путем* уже существует, его содержимое будет перезаписано. Однако если дать параметру *добавлять в файл* значение `true`, Laravel допишет новый вывод в конец этого файла;

- ❐ `appendOutputTo(<полный путь к файлу>)` — аналогичен `sendOutputTo()`, только очередной вывод будет всегда дописываться в конец файла с заданным *путем*;
- ❐ `emailOutputTo()` — высылает вывод по электронной почте по заданному *адресу* или сразу нескольким адресам, указанным в *массиве*:

```
emailOutputTo(<адрес>|<массив адресов>[,
            <только если есть вывод?>=false])
```

Пример:

```
$schedule->call( ... )->hourlyAt(15)
        ->emailOutputTo(['admin@bboard.ru', 'devteam@bboard.ru']);
```

По умолчанию письмо отсылается всегда, даже если задание не сгенерировало никакого вывода (в этом случае письмо будет пустым). Если требуется отсылать

письма, только если задание сгенерировало какой-либо вывод, достаточно параметру *только если есть вывод* дать значение `true`.

В качестве темы отправляемого письма указывается имя, заданное методом `name()` или `description()`, а если имя не задано — строка формата «Scheduled Job Output For <*команда, выполняемая заданием*>». Пример:

```
$schedule->call( ... )->hourlyAt(15)->name('Задание выполнено!')
         ->emailOutputTo(['admin@bboard.ru', 'devteam@bboard.ru']);
```

- `emailWrittenOutputTo(<`*адрес*`>|<`*массив адресов*`>)` — аналогичен `emailOutputTo()`, отправляет письмо, только если задание сгенерирует какой-либо вывод;
- `emailOutputOnFailure()` — высылает вывод по электронной почте, только если задание выполнилось с ошибкой. Формат вызова аналогичен таковому у метода `emailWrittenOutputTo()`.

27.1.4. Исполнение указанного кода перед выполнением задания и после него

Если перед выполнением задания планировщика и (или) после него необходимо исполнить какой-либо код, помогут следующие методы объекта задания:

- `before(<`*анонимная функция*`>)` — исполняет заданную *анонимную функцию* перед выполнением задания. *Анонимная функция* не должна принимать параметры и возвращать результат. Пример:

```
use Illuminate\Support\Facades\Storage;
. . .
$schedule->call( ... )->hourly()->before(function () {
    Storage::disk('local')->append('logs/scheduler.log',
                                   'Приступаю к выполнению задания');
});
```

- `after(<`*анонимная функция*`>)` — исполняет заданную *анонимную функцию* после выполнения задания. *Анонимная функция* не должна принимать параметры и возвращать результат;
- `then()` — то же, что и `after()`;
- `onSuccess(<`*анонимная функция*`>)` — исполняет заданную *анонимную функцию* только после успешного выполнения задания. *Анонимная функция* не должна принимать параметры и возвращать результат;
- `onFailure(<`*анонимная функция*`>)` — исполняет заданную *анонимную функцию*, только если задание выполнилось с ошибкой. *Анонимная функция* не должна принимать параметры и возвращать результат.

27.1.5. Отправка сигналов по указанным интернет-адресам

Иногда требуется сигнализировать каким-либо сторонним интернет-ресурсам о начале и (или) завершении выполнения очередного задания планировщика. Обычно сигналы

такого рода отправляются по протоколу ICMP (Internet Control Message Protocol, протокол межсетевых управляющих сообщений) и схожи с сообщениями, отправляемыми системной утилитой ping.

> ***Для успешной отправки сигналов...***
>
> ...необходимо установить дополнительную библиотеку, набрав команду:
>
> `composer require guzzlehttp/guzzle`

Отправкой сигналов занимаются следующие методы объекта задания:

- `pingBefore(<интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу* перед выполнением задания:

```
$schedule->call( ... )->hourly()
                       ->pingBefore('remote-admin.bboard.ru');
```

- `thenPing(<интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу* после выполнения задания;
- `pingBeforeIf(<условие>, <интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу* перед выполнением задания, если заданное *условие* в результате вычисления дает `true`:

```
$schedule->call( ... )->hourly()
                       ->pingBeforeIf(config('app.sendPings'),
                                      'remote-admin.bboard.ru');
```

- `thenPingIf(<условие>, <интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу* после выполнения задания, если заданное *условие* в результате вычисления дает `true`;
- `pingOnSuccess(<интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу* только после успешного выполнения задания;
- `pingOnFailure(<интернет-адрес>)` — отправляет сигнал по указанному *интернет-адресу*, только если задание выполнилось с ошибкой.

27.2. Работа с заданиями планировщика

27.2.1. Запуск планировщика заданий

27.2.1.1. Запуск с использованием штатного планировщика операционной системы

Как правило, планировщик заданий Laravel заносится в штатный планировщик операционной системы в качестве задания, которое:

- выполняется ежеминутно;
- исполняет команду, запускающую на выполнение процесс планировщика заданий Laravel:

```
php artisan schedule:run [--quiet]
```

  По умолчанию при выполнении очередного задания планировщик выводит сообщение в командной строке. Ключ `--quiet` предписывает не делать этого.

Способы добавления нового задания в системный планировщик описываются в документации по операционной системе. Автор книги, пользующийся Microsoft Windows 10, при создании задания указал следующие ключевые параметры:

- в настройках триггера — события, запускающего выполнение задания (рис. 27.1):
  - в группе переключателей, задающих частоту выполнения, — установил переключатель **Однократно**;
  - установил флажок **Повторять задачу каждые** и выбрал в расположенном правее раскрывающемся списке пункт **1 мин**;
  - выбрал в раскрывающемся списке **в течение** пункт **Бесконечно**;

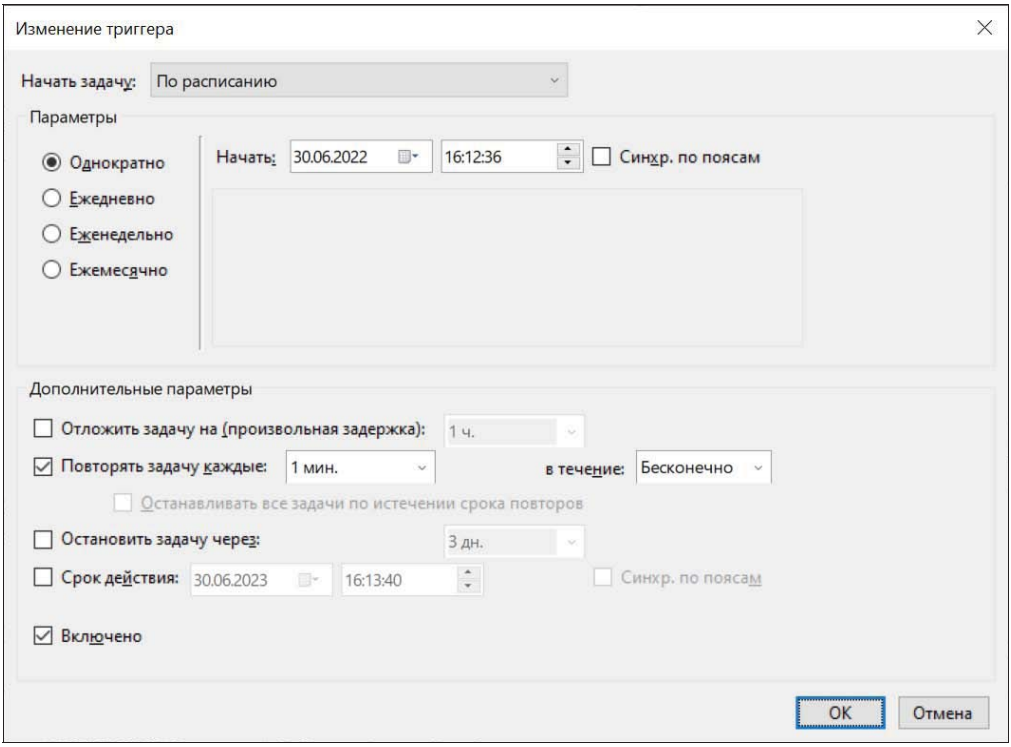


Рис. 27.1. Настройки триггера задания, запускающего планировщик заданий Laravel

- в настройках действия — одной из операций, выполняемых заданием (рис. 27.2):
  - выбрал в раскрывающемся списке **Действие** пункт **Запуск программы**;
  - занес в поле ввода **Программа или сценарий** строку `php` (имя файла исполняющей среды этой платформы);
  - занес в поле ввода **Добавить аргументы (необязательно)** строку `artisan schedule:run` (имена утилиты artisan и команды, запускающей планировщик Laravel);
  - занес в поле **Рабочая папка (необязательно)** полный путь к папке проекта.

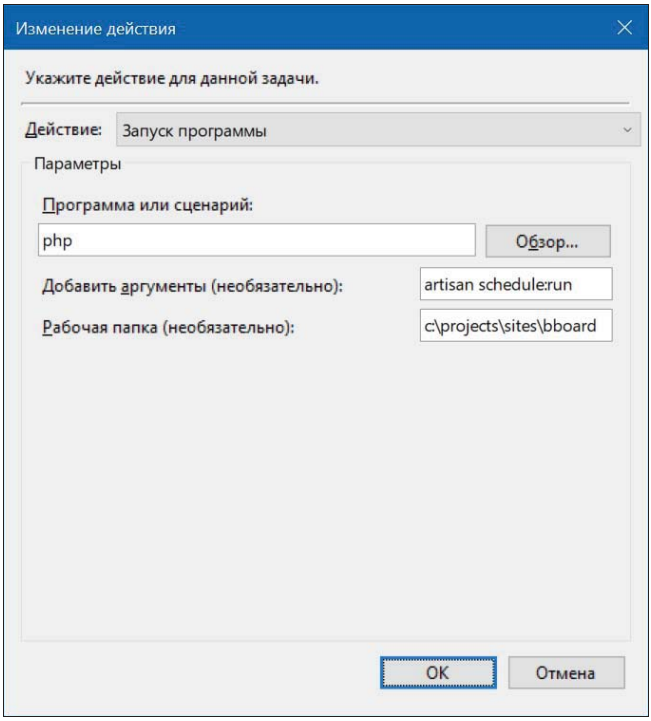


Рис. 27.2. Настройки действия задания, запускающего планировщик заданий Laravel

27.2.1.2. Запуск в независимом режиме

Если по какой-либо причине штатным планировщиком системы воспользоваться невозможно, планировщик заданий Laravel можно запустить в независимом режиме. Для этого достаточно отдать команду:

```
php artisan schedule:work [--quiet]
```

Назначение командного ключа `--quiet` описывалось в *разд. 27.2.1.1*.

Запущенный таким образом планировщик будет работать постоянно, каждую минуту перебирая список созданных заданий, чтобы выяснить, не пора ли запустить какое-либо их них. Завершить его исполнение можно нажатием комбинации клавиш <Ctrl>+<C> или <Ctrl>+<Break>.

Поскольку, будучи запущенным в независимом режиме, планировщик Laravel работает постоянно, он отнимает больше системных ресурсов, чем при запуске из системного планировщика. Поэтому при эксплуатации сайта не рекомендуется использовать независимый режим.

27.2.2. Вывод списка заданий планировщика

Для вывода списка заданий планировщика, созданных к текущему времени, применяется команда:

```
php artisan schedule:list
```

Список заданий состоит из двух столбцов:

- описание задания: слово **Closure** — если задание создано на основе анонимной функции вызовом метода `call()`, команда утилиты artisan, операционной системы или путь к классу отложенного задания — если задание было создано на их основе;
- промежуток времени, спустя который задание будет запущено в следующий раз.

27.2.3. Удаление распределенных блокировок

Для предотвращения наложения заданий и запуска заданий одновременно на нескольких серверах фреймворк использует распределенные блокировки. Может случиться так, что из-за программного сбоя какая-либо из блокировок не снимается («вечная» блокировка), в результате чего соответствующее ей задание никак не может запуститься.

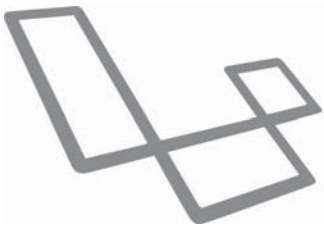
В таком случае может помочь принудительное удаление всех распределенных блокировок, используемых планировщиком заданий. Оно выполняется командой:

```
php artian schedule:clear-cache
```

27.3. События, генерируемые при выполнении заданий планировщика

В процессе выполнения заданий планировщика генерируются следующие события, классы которых объявлены в пространстве имен `Illuminate\Console\Events`:

- `ScheduledTaskStarting` — генерируется перед выполнением задания планировщика. Поддерживается свойство `task`, хранящее объект выполняемого задания;
- `ScheduledTaskFinished` — генерируется после выполнения задания планировщика. Поддерживаются свойства:
  - `task` — объект выполненного задания;
  - `runtime` — продолжительность выполнения задания в секундах в виде вещественного числа, округленная до сотых;
- `ScheduleBackgroundTaskFinished` — генерируется после выполнения задания планировщика, запущенного в фоновом режиме. Поддерживается свойство `task`, хранящее объект выполненного задания;
- `ScheduledTaskSkipped` — генерируется, если задание не должно быть выполнено из-за того, что анонимная функция, указанная в вызове метода `when()`, вернула `false`, или анонимная функция из вызова метода `skip()` вернула `true` (оба метода описаны в *разд. 27.1.2.2*). Поддерживается свойство `task`, хранящее объект невыполненного задания;
- `ScheduledTaskFailed` — генерируется, если задание не было выполнено из-за ошибки. Поддерживаются свойства:
  - `task` — объект невыполненного задания;
  - `exception` — объект исключения, возникшего в задании.



ГЛАВА 28

Локализация

Локализация — это адаптация сайта к какому-либо другому языку, называемому *целевым*. Она заключается прежде всего в переводе веб-страниц с *изначального* языка на целевой.

Laravel позволяет наделить сайт поддержкой произвольного количества языков и дать посетителям возможность переключаться между ними.

Изначальный язык задается в рабочей настройке `app.locale`. В настройке `app.fallback_locale` можно указать язык, на который сайт будет переключаться, если выбранный посетителем язык не поддерживается. По умолчанию обе настройки имеют значение `'en'` (английский).

Если сайт пишется на русском языке, следует указать в качестве изначального языка русский, дав обеим настройкам значение `'ru'`.

Все программные модули, содержащие перевод надписей с изначального языка на целевые (*языковые модули*, или *модули локализации*), должны содержаться в папке lang. В предыдущих версиях Laravel для этого предназначалась папка resources\lang.

28.1. Быстрая локализация

При быстрой локализации для каждого из целевых языков создается отдельный языковый модуль, в котором записываются строки на изначальном языке и те же строки, переведенные на один из целевых языков. Однако быстрая локализация дает возможность локализовать лишь надписи, выводимые на веб-страницах.

Чтобы перевести сайт на один из целевых языков путем быстрой локализации, следует:

1. В папке lang — создать языковый модуль с расширением json и именем, совпадающим с наименованием целевого языка. Так, для перевода русскоязычного сайта на английский язык нужно создать модуль lang\en.json;
2. В созданном на *шаге 1* модуле — записать JSON-код, создающий простой объект JavaScript с набором свойств. Именем отдельного свойства этого объекта станет надпись на изначальном языке, а значением свойства — та же надпись, переведенная на целевой язык.

   Пример JSON-модуля с переводом заголовков столбцов списка объявлений с изначального русского языка на целевой английский:

```
{
    "Товар": "Good",
    "Цена, руб.": "Price, RUR",
    "Автор": "Author",
    "Опубликовано": "Published at"
}
```

3. В коде шаблонов — вместо надписей на изначальном языке вставить вызовы функции `__()` (два символа подчеркивания):

   `__(<надпись на изначальном языке>)`

   В качестве результата эта функция вернет перевод указанной *надписи*, взятый из соответствующего текущему языку JSON-модуля. Если переведенная надпись не будет найдена, функция вернет указанную *надпись*. Пример:

```
<thead>
    <tr>
        <th>{{ __('Товар') }}</th>
        <th>{{ __('Цена, руб.') }}</th>
        <th>{{ __('Автор') }}</th>
        <th>{{ __('Опубликовано') }}</th>
    </tr>
</thead>
```

   Вместо функции `__()` можно использовать полностью аналогичную функцию `trans()`.

   И наконец, доступна для применения директива шаблонизатора `@lang`:

   `@lang(<строка на изначальном языке>)`

   Пример:

```
<th>@lang('Товар')</th>
<th>@lang('Цена, руб.')</th>
```

Для проверки, на нужном ли языке выводятся надписи, можно временно сменить изначальный язык сайта, указанный в настройке `app.locale`, на `en`. Как реализовать переключение текущего языка пользователями, будет рассказано далее.

28.2. Локализация с применением обозначений

При локализации *с применением обозначений* для каждого из поддерживаемых сайтом языков, как целевых, так и изначального, создается своя папка. Она будет хранить языковые PHP-модули со строками, записанными на соответствующем языке. Каждой такой строке дается краткое обозначение, по которому ее впоследствии можно будет извлечь для вывода на экран.

Локализация с применением обозначений позволяет перевести не только надписи на страницах, но и строки, выводимые программно (например, сообщения валидаторов об ошибках ввода и всплывающие сообщения).

Чтобы выполнить локализацию сайта с применением обозначений, следует:

1. В папке lang — для каждого поддерживаемого языка, как целевых, так и изначального, — создать папку для хранения языковых PHP-модулей со строками на этом языке. Папка должна иметь имя, совпадающее с наименованием языка. Так, если русскоязычный сайт переводится на английский язык, следует создать папки lang\ru и lang\en;
2. В созданной на *шаге 1* папке — создать PHP-модуль для хранения строк. Ему можно дать любое имя, за исключением: auth, pagination, passwords и validation (потому что модули с такими именами хранят переведенные строки для контроллеров, встроенных в Laravel, и валидаторов). Сами разработчики Laravel рекомендуют назвать этот модуль messages.php;
3. В созданном на *шаге 2* модуле — записать код, возвращающий в качестве результата ассоциативный массив с переведенными строками. Ключами элементов этого массива станут обозначения строк, которые должны быть уникальными в пределах этого модуля и могут быть выбраны произвольно. Значениями элементов должны быть сами переведенные строки.

   Пример модулей, которые содержат четыре строки, переведенные на русский (изначальный язык сайта) и английский языки:

```
// lang\ru\messages.php
return [
    'main' => 'Главная',
    'rubrics' => 'Рубрики',
    'ads' => 'Объявления',
    'ad_added' => 'Объявление добавлено'
];

// lang\en\messages.php
return [
    'main' => 'Main',
    'rubrics' => 'Rubrics',
    'ads' => 'Adverts',
    'ad_added' => 'Advert has added'
];
```

4. В коде сайта — вместо строк на изначальном языке вставить вызовы функции __() или trans(), указав в качестве параметров строки формата:

   <имя PHP-модуля со строками>.<обозначение переведенной строки>

   Примеры:

```
<h1>{{ __('messages.ads') }}</h1>

public function update($request, Bb $bb) {
    . . .
    $request->session()->flash('success', __('messages.ad_added'));
    . . .
}
```

Помимо модуля messages.php в папке lang\<*обозначение языка*> можно создать следующие модули:

- auth.php — хранит переведенные всплывающие сообщения, выводимые контроллером, который реализует вход на сайт и выход с него;
- pagination.php — хранит переведенные надписи для гиперссылок перехода между частями пагинатора;
- passwords.php — хранит переведенные всплывающие сообщения, выводимые контроллерами, которые реализуют сброс пароля;
- validation.php — хранит переведенные сообщения об ошибках ввода, выводимые встроенными во фреймворк валидаторами.

Если сайт достаточно велик, в папке lang\<*обозначение языка*> для хранения выводимых надписей можно создать несколько языковых модулей. Например, надписи, выводимые на страницах перечня объявлений, можно поместить в модуль bbs.php, надписи со страниц административного раздела — в модуль admin.php и пр.

В каждом вновь созданном проекте уже присутствует папка lang\en, так что сайт создается, можно сказать, уже переведенным на английский язык.

28.2.1. Подстановка параметров в переведенные строки

Есть возможность помещать в заданные места переведенных строк произвольные значения. Она может пригодиться, например, при выводе именных приветствий, инструкций по вводу данных и пр. Для этого следует:

1. Открыть PHP-модуль с переведенными строками на нужном языке (например, модуль lang\ru\messages.php — со строками на русском языке);
2. Поместить в нужное место нужной переведенной строки литерал формата :<*произвольное имя*>, вместо которого будет подставлено заданное значение:

```
return [
    . . .
    'offer_count' => 'Всего :count предложений(е, я)',
];
```

3. Использовать для вывода расширенный формат функции __():

```
__(<обозначение>, <массив с выводимыми значениями>)
```

В заданном ассоциативном *массиве с выводимыми значениями* ключи элементов должны соответствовать именам вставленных в строки на *шаге 2* литералов, а значения элементов будут выведены вместо этих литералов. Пример:

```
<p>{{ __('messages.offer_count', ['count' => 4]) }}</p>
```

Аналогичный расширенный формат поддерживается и функцией trans().

28.2.2. Вывод существительных во множественном числе

При локализации часто возникают затруднения с выводом существительных во множественном числе. Для такого случая Laravel предоставляет встроенные средства выбора

той формы существительного, которая подходит к конкретному случаю. Для их использования следует:

1. Добавить в модуль переведенную строку одного из следующих форматов:

   - *<форма единственного числа>|<форма множественного числа>* — если существительное имеет всего две *формы*: для единственного и множественного чисел (так бывает в английском языке):

     ```
     return [
         . . .
         'offer' => 'offer|offers',
     ];
     ```

   - *<форма 1>|<форма 2>| ... |<форма n>* — если существительное имеет несколько *форм* множественного числа, в зависимости от количества (что характерно для славянских языков). Отдельная *форма* записывается в формате *<количество> <собственно существительное>*, где в качестве *количества* можно указать:

     - *{<n>}* — количество равно *n* (задается в виде целого числа);
     - *[<m>,<n>]* — количество в диапазоне от *m* до *n* включительно. Если в качестве *m* или *n* указать звездочку (*), соответствующая граница диапазона не будет ограничена.

     Пример:

     ```
     return [
         . . .
         'offer' => '{0} Нет предложений|{1} Одно предложение|' .
                    '[2,5] 2-5 предложений|[6,*] Более 5 предложений',
     ];
     ```

     В строках такого рода также можно использовать литералы *:<произвольное имя>* для вывода произвольных значений (см. *разд. 28.2.1*):

     ```
     return [
         . . .
         'offer2' => '{0} :count предложений|' .
                     '{1} :count предложение|' .
                     '[2,4] :count предложения|' .
                     '[5,*] :count предложений',
     ];
     ```

2. Для вывода существительного в нужной форме множественного числа — использовать функцию `trans_choice()`:

   ```
   trans_choice(<обозначение>, <количество>[,
                               <массив с выводимыми значениями>=[]])
   ```

   Примеры:

   ```
   <p>{{ trans_choice('messages.offer', $offerCount) }}</p>
   <p>{{ trans_choice('messages.offer2', $offerCount,
                                         ['count' => $offerCount]) }}</p>
   ```

28.2.3. Локализация сообщений об ошибках ввода

Локализуемые сообщения об ошибках ввода следует хранить в PHP-модулях validation.php. Как и остальные модули с переведенными строками, они должны содержать объявление ассоциативного массива. Ключи элементов этого массива соответствуют именам правил валидации (см. *разд. 10.2.3* и *18.4.1*), а значения элементов могут быть:

- переведенными сообщениями об ошибках — если одно и то же сообщение должно выдаваться во всех случаях:

```
'email' => ':attribute должен быть адресом электронной почты.'
```

- вложенными ассоциативными массивами — если выдаваемое сообщение об ошибке зависит от типа значения, заносимого в элемент управления. Ключи элементов вложенного массива представляют типы заносимого значения: `numeric` (число), `file` (файл), `string` (строка) или `array` (массив). Значения элементов представляют сами сообщения об ошибках ввода. Пример:

```
'max' => [
    'numeric' => 'Число :attribute не должно быть больше :max.',
    'file' => 'Файл :attribute не должен быть больше :max Кбайт.',
    'string' => 'Строка :attribute не должна быть длиннее ' .
                ':max символов.',
    'array' => 'Массив :attribute не должен содержать более ' .
               ':max элементов.'
],
```

Помимо этого, массив из модуля validation.php может содержать следующие элементы:

- `attributes` — ассоциативный массив названий, выводящихся в составе сообщений об ошибках вместо наименований элементов управления (места, где они выводятся, в строках сообщений помечаются литералами `:attribute`, подробности — в *разд. 10.2.4*). Ключи элементов этого массива соответствуют наименованиям элементов управления, а значения зададут выводящиеся названия. Пример:

```
'attributes' => ['title' => 'Название товара',
                 'content' => 'Содержание объявления',
                 'price' => 'Цена'],
```

- `values` — ассоциативный массив строк, которые будут выводиться в составе сообщений об ошибках вместо определенных значений, введенных в элементы управления (место вывода такого значения помечается литералом `:value`, подробности — в *разд. 10.2.4*). Ключи элементов этого массива соответствуют значениям, а значения зададут выводящиеся вместо них строки.

  В качестве примера можно привести набор переключателей, задающий тип объявления и содержащий переключатели со значениями (заданными в атрибутах `value` тегов `<input>`) `buy`, `sell` и `exchange`. Чтобы вместо значений этих переключателей выводились соответственно строки «Покупка», «Продажа» и «Обмен», следует задать в элементе `values` следующий массив:

```
'values' => ['buy' => 'Покупка', 'sell' => 'Продажа',
                                  'exchange' => 'Обмен'],
```

- `custom` — перечень специфических сообщений об ошибках, выводящихся у определенных элементов управления. Значением этого элемента должен быть ассоциативный массив, ключи элементов которого представляют наименования элементов управления, а значениями должны быть вложенные ассоциативные массивы. Ключи элементов вложенных массивов должны соответствовать именам правил валидации, а значения зададут сами сообщения об ошибках. Пример:

```
'custom' => [
    'title' => [
        'require' => 'Введите название товара',
        'max' => 'Длина названия товара превышает :max символов',
    ],
    'price' => [
        'require' => 'Укажите цену товара',
        'numeric' => 'Цена товара должна быть числом',
    ],
],
```

Разумеется, в модуль validation.php можно добавить сообщения об ошибках, которые будут выводиться валидаторами, написанными разработчиками сайта.

Записав в этом модуле все сообщения об ошибках, можно не указывать их в программном коде, выполняющем валидацию данных, и в классах формальных запросов, что упростит разработку.

28.3. Реализация переключения на другой язык

Если сайт поддерживает несколько языков, необходимо дать посетителям возможность переключаться на нужный им язык. Для этого следует:

1. Создать маршрут, указывающий на действие контроллера, которое будет менять текущий язык сайта:

```
Route::get('/setlocale/{locale}',
           [MainController::class, 'setLocale'])
    ->name('setlocale');
```

   Проблема в том, что инструменты Laravel, используемые для переключения на другой язык, позволяют указать язык только на время обработки текущего запроса. Как только поступит следующий запрос, сайт снова переключится на изначальный язык, указанный в настройке `app.locale`.

   Для выхода из такого положения можно сохранить выбранный посетителем язык в серверной сессии и выполнять переключение на него при поступлении каждого запроса в специально написанном посреднике;

2. Создать действие контроллера, которое будет сохранять выбранный посетителем язык в сессии под именем, например, `user_locale`:

```
class MainController extends Controller {
    . . .
    public function setLocale($locale) {
        session(['user_locale' => $locale]);
```

```
            return back();
        }
    }
```

После сохранения выбранного языка нужно выполнить перенаправление на предыдущую страницу — тогда у посетителя создастся впечатление, что просматриваемая им страница была моментально переведена на выбранный им язык;

3. Написать посредник, переключающий сайт на выбранный посетителем язык, назвав его, скажем, `Localize`:

```
use Illuminate\Support\Facades\App;
class Localize {
    public function handle(Request $request, Closure $next) {
        $locale = session('user_locale', config('app.locale'));
        App::setLocale($locale);
        return $next($request);
    }
}
```

Обозначение языка извлекается из серверной сессии или, если его там нет, из рабочей настройки `app.locale`.

Для переключения на выбранный язык используется метод `setLocale(<`*обозначение языка*`>)` фасада `Illuminate\Support\Facades\App`;

4. Занести написанный на *шаге 3* посредник в группу `web` модуля `App\Http\Kernel` после посредника `Illuminate\Session\Middleware\StartSession` (который создает серверную сессию):

```
class Kernel extends HttpKernel {
    . . .
    protected $middlewareGroups = [
        'web' => [
            . . .
            \Illuminate\Session\Middleware\StartSession::class,
            . . .
            \App\Http\Middleware\Localize::class,
        ],
        . . .
    ];
    . . .
}
```

5. Создать гиперссылки для переключения между языками:

```
<a href="{{ route('setlocale', ['locale' => 'en']) }}">English</a>
<a href="{{ route('setlocale', ['locale' => 'ru']) }}">Русский</a>
```

Фасад `App` также поддерживает два метода, которые могут оказаться полезными:

- `getLocale()` — возвращает обозначение текущего языка;
- `isLocale(<`*обозначение языка*`>)` — возвращает `true`, если текущим является язык с указанным *обозначением*, и `false` — в противном случае:

```
@if (App::isLocale('ru'))
    <a href="{{ route('setlocale', ['locale' => 'en']) }}">English</a>
@else
    <a href="{{ route('setlocale', ['locale' => 'ru']) }}">Русский</a>
@endif
```

28.4. Библиотека Laravel Lang: локализация на множество языков

Если нет желания создавать модули локализации для всех поддерживаемых сайтом языков, можно прибегнуть к помощи библиотеки Laravel Lang. Она содержит готовые модули локализации на несколько десятков языков, включая русский:

- JSON-модули быстрой локализации — с надписями для страниц регистрации, входа, проверки адреса электронной почты и сброса пароля, создаваемых командой `ui:auth`;
- PHP-модули локализации с применением обозначений — хранящие всплывающие сообщения и сообщения об ошибках ввода, отображаемые на этих страницах.

> ***Полная документация по Laravel Lang...***
> ...находится по адресу: **https://laravel-lang.com/**.

Для установки библиотеки необходимо набрать команду:

```
composer require laravel-lang/lang --dev
```

Далее следует скопировать необходимые языковые модули из состава этой библиотеки и поместить копии в папку lang проекта. Это позволит сделать еще одна дополнительная библиотека — LaravelLang. Она добавляет утилите artisan поддержку команды, копирующей языковые модули.

Библиотека LaravelLang устанавливается командой:

```
composer require arcanedev/laravel-lang --dev
```

> ***Полная документация по LaravelLang...***
> ...находится по адресу: **https://github.com/ARCANEDEV/LaravelLang**.

Для копирования языковых модулей языка с указанным *обозначением* в состав проекта служит команда:

```
php artisan tranc:publish <обозначение языка> [--json] [--force]
```

По умолчанию копируется только соответствующая заданному языку папка с PHP-модулями, реализующими локализацию с применением обозначений. Если какой-либо языковый модуль уже присутствует в составе проекта, он не копируется.

Поддерживаются следующие командные ключи:

- `--json` — скопировать также JSON-модуль быстрой локализации;
- `--force` — принудительно копировать модули, даже если их копии уже присутствуют в составе проекта.

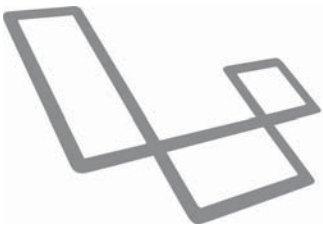
К сожалению, эта команда имеет недостатки. Во-первых, она помещает в создаваемую папку с PHP-модулями также и совершенно не нужный JSON-модуль быстрой локализации, представляющий собой копию JSON-модуля, помещаемую непосредственно в папку lang. Во-вторых, в папку с PHP-модулями также копируется ненужный модуль validation-nova.php[1]. Эти модули можно удалить.

Скопировать языковые модули можно и вручную, выполнив следующие действия:

1. Создать в папке lang папку с именем, совпадающим с обозначением нужного языка;
2. Открыть папку vendor\laravel-lang\lang\locales, содержащую JSON-модули быстрой локализации;
3. Найти в этой папке модуль с именем вида <*обозначение нужного языка*>.json и скопировать его в папку lang;
4. Открыть папку vendor\laravel-lang\lang\locales\<*обозначение нужного языка*> с PHP-модулями локализации с применением обозначений;
5. Найти в этой папке модули auth.php, pagination.php, passwords.php и validation.php и скопировать их в папку lang\<*обозначение нужного языка*>.

[1] Судя по всему, он содержит сообщения, выводимые дополнительной библиотекой Laravel Nova (**https://nova.laravel.com/**), которая реализует административную панель и не описывается в этой книге.

ГЛАВА 29



Кеширование

Laravel предоставляет базовые средства кеширования данных на стороне сервера, позволяющие сохранять в кеше произвольные данные (элементарные значения, массивы, коллекции записей и др.). Однако кешировать веб-страницы целиком или их фрагменты невозможно, по крайней мере, встроенными средствами.

В то же время Laravel может управлять кешированием страниц на стороне клиента, задавая в отсылаемых ответах соответствующие заголовки.

29.1. Кеширование на стороне сервера

29.1.1. Подготовка подсистемы кеширования

29.1.1.1. Настройка подсистемы кеширования

Настройки подсистемы кеширования на стороне сервера записаны в модуле config\cache.php:

- `stores` — ассоциативный массив служб, в которых будут храниться кешируемые данные. Ключи элементов массива задают имена служб, а значениями являются вложенные ассоциативные массивы с параметрами отдельных служб. В этих массивах можно указать такие параметры:
  - `driver` — тип службы. Поддерживаются значения:
    - `file` — обычные файлы в заданной папке;
    - `database` — таблица в реляционной базе данных;
    - `redis` — нереляционная база данных Redis;
    - `memcached` — сервер кеширования Memcached;
    - `dynamodb` — нереляционная база данных Amazon DynamoDB;
    - `apc` — PHP-расширение Alternative PHP User Cache;
    - `array` — массив в оперативной памяти;
    - `null` — данные фактически нигде не кешируются. Используется только при отладке.

Следующая настройка используется только службой `file`:

- `path` — полный путь к папке, в которой будут храниться файлы с кешируемыми данными (по умолчанию: полный путь к папке storage\framework\cache\data).

Следующие настройки используются только службами `database` и `redis`:

- `connection` — имя базы данных из указанных в настройках `database.connections` (см. *разд. 3.4.2.4*) или `redis.connections` (см. *разд. 25.1.3*). По умолчанию: `null` (база данных по умолчанию);
- `lock_connection` — имя базы данных, в которой будет храниться список распределенных блокировок (описаны в *разд. 29.1.3*). Если указано значение `null`, будет использована база данных по умолчанию. Значения этой настройки по умолчанию у разных служб: `null` и `'default'`, дающие одинаковый эффект.

Следующая настройка используется только службой `database`:

- `table` — имя таблицы, в которой будет храниться кеш (по умолчанию: `cache`).

Следующие настройки используются только службой `memcached`:

- `servers` — перечень используемых серверов Memcached. Указывается в виде массива, каждый элемент которого представляет собой ассоциативный массив с параметрами отдельного сервера. Ключи элементов такого вложенного массива соответствуют отдельным параметрам, а значения элементов задают значения этих параметров. Поддерживаются следующие параметры серверов:
  - `host` — интернет-адрес сервера. Значение берется из локальной настройки `MEMCACHED_HOST`, изначально отсутствующей в файле .env. По умолчанию: **127.0.0.1**;
  - `post` — номер TCP-порта, через который работает сервер. Значение берется из локальной настройки `MEMCACHED_PORT`, изначально отсутствующей в файле .env. По умолчанию: 11211;
  - `weight` — вероятность, с которой тот или иной сервер будет выбран для использования, в виде целого числа от 0 до 100 (по умолчанию: 100).

  Изначально в этом массиве присутствует лишь один сервер;
- `persistent_id` — уникальный идентификатор устойчивого экземпляра. Значение берется из локальной настройки `MEMCACHED_PERSISTENT_ID`, изначально отсутствующей в файле .env;
- `sasl` — учетные записи для входа на сервер Memcached, если таковой требует их указания. Значение представляет собой массив с двумя элементами: именем пользователя и паролем. Значения элементов этого массива берутся из локальных настроек соответственно `MEMCACHED_USERNAME` и `MEMCACHED_PASSWORD`, изначально отсутствующих в файле .env;
- `options` — дополнительные параметры для подключения к серверу Memcached. Задаются в виде ассоциативного массива, ключи элементов которого соответствуют отдельным параметрам, а значения элементов задают значения этих параметров. Описание дополнительных параметров см. в документации по Memcached.

Следующие настройки используются только службой `dynamodb`:

- `key` — ключ доступа. Значение берется из локальной настройки `AWS_ACCESS_KEY_ID`, присутствующей в файле .env, но изначально «пустой»;
- `secret` — секретный ключ. Значение берется из локальной настройки `AWS_SECRET_ACCESS_KEY`, присутствующей в файле .env, но изначально «пустой»;
- `region` — обозначение региона. Значение берется из локальной настройки `AWS_DEFAULT_REGION`. По умолчанию: `us-east-1`;
- `table` — имя таблицы, в которой будет храниться кеш. Значение берется из локальной настройки `DYNAMODB_CACHE_TABLE`, изначально отсутствующей в файле .env. Значение по умолчанию: `cache`;
- `endpoint` — имя используемой точки контроля. Значение берется из локальной настройки `DYNAMODB_ENDPOINT`, изначально отсутствующей.

Следующая настройка используется только службой `array`:

- `serialize` — если `false`, значение будет записываться в кеш в изначальном виде, если `true` — будет предварительно сериализоваться. Если в кеше предстоит хранить сложные структуры данных (например, массивы и объекты), имеет смысл дать этой настройке значение `true`. По умолчанию: `false`.

Изначально присутствуют службы: `file`, `database`, `redis`, `memcached`, `dynamodb`, `apc` и `array`;

- `default` — служба, используемая по умолчанию, если при выполнении доступа к кешу служба не была указана явно. Значение берется из локальной настройки `CACHE_DRIVER`. По умолчанию: `file`;
- `prefix` — префикс, добавляемый к именам кешируемых значений. Используется только службами, способными обслуживать несколько приложений одновременно: `database`, `redis`, `memcached`, `dynamodb` и `apc`. Значение берется из локальной настройки `CACHE_PREFIX`, изначально не существующей. По умолчанию: строка, составленная из названия проекта (берется из локальной настройки `APP_NAME`) и слова `cache`, записанная в стиле snake_case.

29.1.1.2. Создание таблицы для хранения кеша

Если была выбрана служба кеша `database`, т. е. обычная реляционная база данных, в этой базе нужно создать таблицу, в которой и будет храниться кеш.

Миграция, формирующая эту таблицу, создается набором команды:

```
php artisan cache:table
```

Созданная миграция имеет имя формата *<временна́я отметка>*_create_cache_table.php. К сожалению, она в любом случае создает таблицу с именем `cache`. Так что, если в настройке `stores.database.table` было указано другое имя для таблицы кеша, код миграции придется исправить вручную.

Кроме того, эта миграция создает таблицу `cache_locks`, предназначенную для хранения списка распределенных блокировок.

После создания миграции следует выполнить миграции.

29.1.2. Работа с кешем стороны сервера

29.1.2.1. Сохранение данных в кеше и их правка

Проще всего сохранить единственное значение в кеше, вызвав функцию-хелпер `cache(<массив>[, <время хранения>=null])`. Заданный ассоциативный *массив* должен содержать один элемент, ключ которого задаст имя сохраняемого в кеше значения, а его значение — само это значение. В качестве *времени хранения* значения в кеше можно указать:

- собственно время хранения в секундах:

```
cache(['bbs' => $bbs], 120);
```

- момент времени, до которого значение будет храниться в кеше, в виде объекта класса `Carbon` или в любом представлении, поддерживаемом PHP:

```
cache(['rubrics' => $rubrics], now()->addMinutes(5));
```

- `null` — тогда кешируемое значение будет храниться вечно:

```
cache(['counter' => null]);
```

Функция `cache()` возвращает `true`, если значение было успешно записано в кеш, и `false` — в противном случае.

Также можно использовать методы объекта службы кеша — они дают дополнительные возможности и некоторую прибавку в производительности. Объект службы кеша «скрывается» за фасадом `Illuminate\Support\Facades\Cache`, а методы, заносящие данные в кеш, приведены далее:

- `put()` — сохраняет в кеше заданное значение или значения, в зависимости от формата вызова, и возвращает `true` в случае успеха и `false` в случае неудачи:

```
put(<имя значения>, <значение>[, <время хранения>=null])
put(<ассоциативный массив значений>[, <время хранения>=null])
```

  Первый формат сохраняет в кеше одно *значение* под заданным *именем*:

```
use Illuminate\Support\Facades\Cache;
. . .
Cache::put('bbs', $bbs, 120);
```

  Второй формат позволяет сохранить произвольное количество значений. Ключи элементов указанного *массива* зададут имена кешируемых значений, а значения элементов — сами эти значения. Пример:

```
Cache::put(['rubrics' => $rubrics, 'users' => $users],
           now()->addMinutes(5));
```

- `forever(<имя значения>, <значение>)` — сохраняет в кеше *значение* под заданным *именем* навсегда. Возвращаемый результат такой же, как и у метода `put()`. Пример;

```
Cache::forever('counter', 0);
```

- `putMany()` — аналогичен второму формату вызова метода `put()`;
- `add()` — сохраняет в кеше заданное значение под указанным именем только в том случае, если такое значение в кеше отсутствует. Формат вызова совпадает с первым форматом метода `put()`.

Следующие методы кеша позволяют изменить ранее записанные данные:

- `increment(<имя>[, <величина>=1])` — увеличивает значение, хранящееся в кеше под заданным *именем*, на указанную *величину*:

  ```
  Cache::increment('counter');
  ```

- `decrement(<имя>[, <величина>=1])` — уменьшает значение, хранящееся в кеше под заданным *именем*, на указанную *величину*:

  ```
  Cache::decrement('counter', 2);
  ```

Объект службы кеша также можно получить, вызвав функцию `cache()` без параметров:

```
cache()->forever('counter', 0);
```

По умолчанию данные сохраняются в службе кеша, указанной в настройках как используемая по умолчанию. Чтобы сохранить значение в другом кеше, следует вызвать у фасада `Cache` метод `store(<имя службы кеша>)`, а описанные здесь методы вызывать у возвращенного им результата — объекта службы кеша с заданным *именем*. Пример:

```
Cache::store('database')->put('bbs', $bbs, 120);
```

Вместо метода `store()` можно использовать полностью ему аналогичный метод `driver()`.

29.1.2.2. Чтение данных из кеша

Проще всего прочитать кешированное значение с заданным *именем*, вызвав функцию `cache()`:

```
cache(<имя>[, <значение по умолчанию>=null])
```

Если значение с заданным *именем* в кеше отсутствует, будет возвращено *значение по умолчанию*. Пример:

```
$bbs = cache('bbs');
```

Вместо *значения по умолчанию* можно указать анонимную функцию, не принимающую параметры и возвращающую результат, который и станет значением по умолчанию:

```
$bbs = cache('bbs', function () { return Bb::all(); });
```

Следующие методы, поддерживаемые объектом службы кеша, дают расширенные возможности и работают несколько быстрее:

- `get()` — возвращает значение или значения из кеша. Поддерживаются два формата вызова:

  ```
  get(<имя>[, <значение по умолчанию>=null])
  get(<массив имен>)
  ```

  Первый формат полностью аналогичен функции `cache()`:

  ```
  use Illuminate\Support\Facades\Cache;
  . . .
  $bbs = Cache::get('bbs');
  ```

  Второй формат возвращает массив значений с именами, присутствующими в указанном *массиве*. Если какого-либо значения в кеше нет, соответствующий элемент возвращаемого массива будет хранить `null`. Пример:

```
$data = Cache::get(['bbs', 'rubrics']);
$bbs = $data[0];
$rubrics = $data[1];
```

- many() — полностью аналогичен второму формату вызова метода get();
- has(<*имя*>) — возвращает true, если в кеше существует значение с заданным *именем*, не равное null, и false — в противном случае:

```
if (Cache::has('rubrics'))
    // Значение rubrics есть в кеше
```

- missing(<*имя*>) — возвращает true, если в кеше, наоборот, отсутствует значение с заданным *именем*, не равное null, и false — в противном случае;
- remember() — возвращает значение, сохраненное в кеше под заданным *именем*. Если такого значения в кеше нет, записывает под тем же *именем* результат, возвращенный не принимающей параметров *анонимной функцией*. Формат вызова:

```
remember(<имя>, <время хранения>, <анонимная функция>)
```

Пример:

```
$rubrics = Cache::remember('rubrics', 300, function () {
    return Rubric::all();
});
```

- rememberForever(<*имя*>, <*анонимная функция*>) — аналогичен remember(), только сохраняет значение в кеше навсегда;
- sear() — то же самое, что и rememberForever().

29.1.2.3. Удаление данных из кеша

Для удаления данных из кеша применяются следующие методы, поддерживаемые объектом службы кеша:

- pull(<*имя*>[, <*значение по умолчанию*>=null]) — возвращает значение с заданным *именем*, после чего удаляет его из кеша. Если значения с таким *именем* нет, возвращает *значение по умолчанию*. Пример:

```
use Illuminate\Support\Facades\Cache;
. . .
$bbs = Cache::pull('bbs');
```

- forget(<*имя*>) — удаляет из кеша значение с заданным *именем* и возвращает true, если значение было успешно удалено, или false — в противном случае:

```
Cache::forget('bb');
```

- delete() — то же самое, что и forget();
- deleteMultiple(<*массив имен*>) — удаляет из кеша значения с именами, приведенными в указанном *массиве*. Возвращает true, если все значения были успешно удалены, и false — в противном случае;
- flush() — удаляет все значения из кеша;
- clear() — то же самое, что и flush().

Метод `flush()` очищает кеш полностью...

...включая значения, сохраненные другими сайтами (если кеш совместно используется несколькими сайтами).

Утилита artisan предоставляет две команды, позволяющие удалить данные из кеша:

- `php artisan cache:forget <имя значения> [<имя службы кеша>]`

  Удаляет значение с заданным *именем*, хранящееся службой кеша по умолчанию. Чтобы удалить значение, записанное в кеше из другой службы, следует указать *имя* этой *службы*;

- `php artisan cache:clear [<имя службы кеша>]`

  Удаляет все значения из кеша, хранящегося службой по умолчанию. Чтобы очистить кеш из другой службы, следует указать ее *имя*.

29.1.3. Распределенные блокировки

Некоторые критичные действия (например, правка в базе данных записи, хранящей ключевые значения) должны выполняться строго одним процессом. Если такое действие одновременно попытаются выполнить сразу два процесса или более, возможно искажение или даже потеря важных данных.

Обеспечить выполнение подобного рода критичных действий одним процессом можно, применяя *распределенные блокировки*. Процесс, первым начавший выполнять критичное действие, создает и накладывает блокировку, после чего «опоздавшие» процессы будут вынуждены ждать, пока первый процесс не закончит выполнение действия и не снимет блокировку.

Распределенные блокировки реализуются через кеш стороны сервера, хранящийся службой: `memcached`, `dynamodb`, `redis`, `database`, `file` или `array`. Никаких специфических настроек задавать у кеша не нужно.

Laravel сам использует распределенные блокировки для предотвращения наложения заданий планировщика, реализации уникальных отложенных заданий и др.

29.1.3.1. Немедленные распределенные блокировки

Если процесс, собирающийся выполнить критичное действие, не может наложить *немедленную распределенную блокировку* (поскольку это действие уже выполняется другим, более «расторопным» процессом), Laravel сразу же сообщает ему об этом. В таком случае процесс либо откладывает выполнение действия на потом, либо предпринимает какие-либо другие шаги.

Немедленную блокировку можно создать и наложить двумя способами: простым и сложным.

Простой способ пригоден в большинстве случаев. Для его реализации нужно:

1. Создать блокировку — заранее, перед выполнением критичного действия.

   Блокировка создается вызовом метода `lock(<имя блокировки>)` фасада `Cache`. *Имя блокировки* указывается в виде строки и должно быть уникальным. В качестве результата метод возвращает объект блокировки, который нужно сохранить в переменной.

2. Наложить блокировку — непосредственно перед началом выполнения критичного действия.

   Блокировка накладывается вызовом у полученного на *шаге 1* объекта блокировки метода `get(<анонимная функция>)`. Критичное действие реализуется в теле *анонимной функции*, которая не должна принимать параметры и возвращать результат.

   Анонимная функция будет вызвана, только если блокировку удалось наложить, в противном случае метод `get()` ничего не сделает. После выполнения *анонимной функции* блокировка будет автоматически снята.

Пример:

```
use Illuminate\Support\Facades\Cache;
. . .
$bb = Bb::find($bb_id);
$lock = Cache::lock('editing_bb_' . $bb_id);
. . .
$lock->get(function () {
    $bb->price = $bb->price * 0.9;
    $bb->save();
});
```

Сложный способ позволяет более гибко реагировать в случае, если блокировку наложить не удалось. Для его реализации следует:

1. Создать блокировку вызовом метода `lock()` фасада `Cache` в формате:

   `lock(<имя блокировки>, <время существования блокировки>)`

   Время существования блокировки указывается в секундах. Как только оно истечет, блокировка будет автоматически снята — это сделано для предотвращения появления «вечных» блокировок, которые могут нарушить работу сайта.

2. Наложить блокировку вызовом того же метода `get()`, но без параметров. В этом случае метод `get()` возвращает `true`, если блокировку удалось наложить, и `false` — в противном случае.

   Выполнять критичное действие можно лишь в том случае, если блокировку наложить удалось.

3. Снять блокировку — после завершения выполнения критичного действия. Это выполняется вызовом у объекта блокировки метода `release()`. Он возвращает `true`, если снять блокировку удалось, и `false` — в противном случае.

Пример:

```
$bb = Bb::find($bb_id);
$lock = Cache::lock('editing_bb_' . $bb_id, 5);
. . .
if ($lock->get()) {
    $bb->price = $bb->price * 0.9;
    $bb->save();
    $lock->release()
} else {
    // Блокировку наложить не удалось
}
```

29.1.3.2. Распределенные блокировки с ожиданием

Распределенная блокировка с ожиданием отличается от немедленной тем, что в случае, если ее не удалось наложить, Laravel будет ждать в течение заданного времени, пока наложивший блокировку процесс не снимет ее.

Блокировку с ожиданием можно создать и наложить двумя способами: простым и сложным.

Чтобы наложить блокировку с ожиданием простым способом, нужно:

1. Создать блокировку вызовом метода `lock(<имя блокировки>)` фасада `Cache`;
2. Наложить блокировку вызовом у ее объекта метода `block()`:

   `block(<время ожидания>, <анонимная функция>)`

   Время ожидания снятия блокировки другим процессом указывается в секундах. *Анонимная функция* реализует критичное действие и аналогична той, что применяется в вызове метода `get()` (см. *разд. 29.1.3.1*).

   Если блокировка не была снята за указанное время ожидания, будет сгенерировано исключение `Illuminate\Contracts\Cache\LockTimeoutException`.

Пример:

```
use Illuminate\Contracts\Cache\LockTimeoutException;
. . .
$bb = Bb::find($bb_id);
$lock = Cache::lock('editing_bb_' . $bb_id);
. . .
try {
    $lock->block(3, function () {
        $bb->price = $bb->price * 0.9;
        $bb->save();
    });
} catch (LockTimeoutException $e) {
    // Блокировку наложить не удалось
}
```

Для наложения блокировки с ожиданием сложным способом необходимо:

1. Создать блокировку вызовом метода `lock()` фасада `Cache` в формате:

   `lock(<имя блокировки>, <время существования блокировки>)`

2. Наложить блокировку — вызовом у ее объекта метода `block(<время ожидания>)`;
3. Снять блокировку — вызовом у ее объекта метода `release()`.

Пример:

```
$bb = Bb::find($bb_id);
$lock = Cache::lock('editing_bb_' . $bb_id);
. . .
try {
    $lock->block(3);
    $bb->price = $bb->price * 0.9;
```

```
    $bb->save();
    $lock->release();
} catch (LockTimeoutException $e) {
    // Блокировку наложить не удалось
}
```

29.1.3.3. Передача распределенных блокировок между процессами

Иногда бывает так, что распределенная блокировка накладывается в одном процессе (например, в действии контроллера), а снимается в другом (скажем, в отложенном задании). В таком случае следует предпринять следующие шаги:

1. В классе-получателе блокировки — объявить свойство для хранения жетона владельца блокировки. Этот жетон имеет вид строки и идентифицирует процесс, создавший блокировку. Также следует предусмотреть в конструкторе параметр, через который будет передаваться этот жетон. Пример класса отложенного задания, получающего жетон:

```
class BbEditing implements ShouldQueue {
    . . .
    protected $bb;
    // Свойство для хранения жетона владельца блокировки
    protected $owner;
    . . .
    public function __construct($bb, $owner) {
        $this->bb = $bb;
        $this->owner = $owner;
    }
}
```

2. В процессе, наложившем блокировку, — получить жетон владельца уже наложенной блокировки и передать его конструктору объекта-получателя блокировки.

   Жетон владельца можно получить, вызвав у объекта блокировки метод `owner()`:

```
$bb = Bb::find($bb_id);
$lock = Cache::lock('editing_bb_' . $bb_id);
. . .
if ($lock->get())
    BbEditing::dispatch($bb, $lock->owner());
```

3. В объекте-получателе блокировки — извлечь жетон владельца блокировки и с его помощью снова наложить блокировку.

   Повторное наложение блокировки выполняется методом `restoreLock()` фасада `Cache`:

```
restoreLock(<имя блокировки>, <жетон владельца блокировки>)
```

   После этого воссозданную в другом процессе распределенную блокировку можно снять. Пример:

```
class BbEditing implements ShouldQueue {
    . . .
```

```
        public function handle() {
            $bb->price = $bb->price * 0.9;
            $bb->save();
            Cache::restoreLock('editing_bb_' . $this->bb->id,
                               $this->owner)
                 ->release();
        }
    }
```

Laravel также предусматривает возможность снятия блокировки в другом процессе, не зная жетона владельца. Для этого в объекте-получателе блокировки нужно:

1. Создать блокировку с тем же именем — вызовом метода `lock(<имя блокировки>)` фасада `Cache`;
2. Снять только что созданную блокировку и все блокировки с тем же именем — вызовом метода `forceRelease()` объекта блокировки, полученного на *шаге 1*.

Пример:

```
class BbEditing implements ShouldQueue {
    . . .
    public function handle() {
        . . .
        Cache::lock('editing_bb_' . $this->bb->id)->forceRelease();
    }
}
```

Зачем предусмотрены два способа снятия блокировки в другом процессе, непонятно. Насколько удалось выяснить автору, оба они успешно работают и не приводят к каким-либо побочным эффектам.

29.1.4. События, генерируемые кешем

Все службы кеша в процессе работы генерируют следующие события, классы которых объявлены в пространстве имен `Illuminate\Cache\Events`:

- `CacheEvent` — базовый абстрактный класс события кеша. Поддерживается свойство `key`, хранящее имя записываемого, извлекаемого или удаляемого значения.

  Все остальные классы событий кеша являются производными от этого класса;
- `KeyWritten` — генерируется при записи в кеш нового значения. Поддерживаются свойства:
  - `value` — само записанное значение;
  - `seconds` — время хранения значения в секундах;
- `CacheHit` — генерируется, если извлекаемое значение содержится в кеше. Поддерживается свойство `value`, хранящее само извлекаемое значение;
- `CacheMissed` — генерируется, если извлекаемое значение отсутствует в кеше;
- `KeyForgotten` — генерируется после удаления значения из кеша.

29.2. Кеширование на стороне клиента

Для управления кешированием сгенерированных веб-страниц на стороне клиента предназначен посредник `Illuminate\Http\Middleware\SetCacheHeaders`, имеющий обозначение `cache.headers`. Этот посредник следует связать с маршрутами на действия контроллера, генерирующие страницы, у которых нужно указать настройки кеширования на стороне клиента.

У посредника указывается параметр в виде набора отдельных настроек кеширования, разделенных точками с запятой (`;`). Поддерживаются следующие настройки:

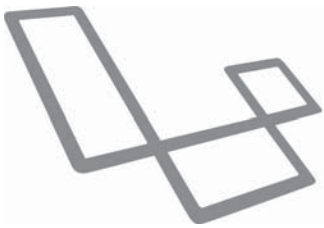
- `etag` — указывает поместить в отправляемый ответ заголовок `ETag`, хранящий идентификатор сгенерированной страницы, в качестве которого выступает ее хеш. Используется веб-обозревателем, чтобы выяснить, изменилась ли страница после ее последнего посещения;
- `last_modified=<`*время*`>` — указывает поместить в отправляемый ответ заголовок `Last-Modified`, задающий время последнего изменения страницы. *Время* можно указать в любом формате, поддерживаемом PHP;
- любые значения, указываемые в заголовке `Cache-Control`. Если в состав значения входит дефис, в параметре посредника `SetCacheHeaders` вместо дефиса следует указать символ подчеркивания (например, значение `max-age` должно быть записано как `max_age`).

Примеры:

```
Route::get('/', [MainController::class, 'index'])
     ->middleware('cache.headers:public;etag');

// Запрещаем кеширование страницы
Route::get('/news', [MainController::class, 'news'])
     ->middleware('cache.headers:no_cache;no_store;must_revalidate');

// Указываем в качестве времени последнего изменения главной страницы
// время правки наиболее "свежего" объявления
Route::get('/', [MainController::class, 'index']));
. . .
class MainController extends Controller {
    public function __construct(Request $request) {
        if ($request->routeIs('index')) {
            $lastBb = Bb::latest('updated_at')->limit(1)->first();
            if ($lastBb)
                $this->middleware('cache.headers:last_modified=' .
                                  $lastBb->updated_at);
    }
    . . .
}
```



ГЛАВА 30

Разработка веб-служб

Веб-служба — это серверная программа, выдающая не обычную веб-страницу, а данные в каком-либо компактном формате, обычно JSON. Совокупность веб-служб образует *бэкенд* (или серверную часть).

Данные, оправляемые бэкендом, на стороне клиента обрабатываются и выводятся на страницы *фронтендом* (или клиентской частью), реализованным в виде страницы с набором веб-сценариев.

Веб-службы строятся по принципам REST (Representational State Transfer, передача состояния представления), согласно которым:

- обрабатываемый фрагмент данных идентифицируется его интернет-адресом (например, чтобы получить список рубрик, следует обратиться по интернет-адресу **/api/rubrics/**, а чтобы получить сведения о рубрике с ключом 11 — по интернет-адресу **/api/rubrics/11/**);
- действие, выполняемое с фрагментом данных, идентифицируется HTTP-методом (так, для загрузки рубрики применяется метод GET, для добавления рубрики — метод POST, для правки — метод PUT или PATCH, а для удаления — метод DELETE);
- состояние клиента хранится на стороне клиента.

30.1. Бэкенды: преобразование данных — базовые инструменты

Веб-службы программируются с применением тех же инструментов, что и традиционные сайты, — маршрутов, контроллеров, моделей, посредников и др.

Маршруты, ведущие на бэкенд (API-маршруты), следует записывать в модуле routes\api.php. Ко всем этим маршрутам автоматически добавляется префикс **/api**.

Действия контроллеров, входящих в состав бэкенда, должны отправлять фронтенду данные в формате JSON, представляющие собой нотацию объектов языка JavaScript и называемые *JSON-объектами*.

Чаще всего фронтенду отправляются либо отдельные записи моделей, либо коллекции этих записей. В первом случае JSON-объект содержит набор свойств, хранящих значе-

ния отдельных полей записи, объекты связанных записей первичных моделей или коллекции связанных записей вторичных моделей. Во втором случае JSON-объект чаще всего содержит свойство, традиционно называемое `data` и содержащее массив объектов записей.

Базовые инструменты, кодирующие записи моделей в формат JSON, встроены непосредственно в базовый класс модели. Они позволяют сгенерировать JSON-объект, включающий либо все поля записи, либо лишь указанные разработчиком, а также добавить в JSON-объект свойства, значения которых вычисляются программно.

30.1.1. Выдача данных в формате JSON

Чтобы отправить клиенту данные в формате JSON, следует выполнить любую из следующих манипуляций:

- вернуть из действия контроллера результат вызова метода `toJson([<параметры кодирования>=0])`. Этот метод можно вызвать как у коллекции объектов модели:

```
// API-маршруты записываются в модуле routes\api.php
Route::get('/rubrics', [ApiRubricController::class, 'index']);
. . .
class ApiRubricController extends Controller {
    public function index() {
        return Rubric::all()->toJson(JSON_UNESCAPED_UNICODE);
    }
    . . .
}
. . .
// При запросе по пути /api/rubrics/ (не забываем, что ко всем путям
// в API-маршрутах автоматически добавляется префикс /api) будет выдан
// следующий ответ:
[
    {
        "id": 1, "name": "Здания", "parent_id": null,
        "created_at": "2022-05-06T15:10:13.000000 Z",
        "updated_at": "2022-05-23T15:50:11.000000Z"
    },
    {
        "id": 2,"name": "Дома", "parent_id": "1",
        "created_at": "2022-05-06T15:13:47.000000Z",
        "updated_at": "2022-05-06T15:13:47.000000Z"
    },
    . . .
]
```

  так и у отдельного объекта модели:

```
Route::get('/rubrics/{rubric}', [ApiRubricController::class, 'show']);
. . .
class ApiRubricController extends Controller {
    . . .
```

```
    public function show(Rubric $rubric) {
        return $rubric->toJson(JSON_UNESCAPED_UNICODE);
    }
    . . .
}
. . .
// При запросе по пути /api/rubrics/11/ будет выдан следующий ответ:
{
    "id": 11, "name": "Бытовая", "parent_id": "10",
    "created_at": "2022-05-16T16:13:08.000000Z",
    "updated_at": "2022-05-16T16:24:59.000000Z"
}
```

В вызове метода `toJson()` можно указать *параметры кодирования* в формат JSON, поддерживаемые встроенной в PHP функцией `json_encode()`. Так, использованная в приведенных примерах опция кодирования `JSON_UNESCAPED_UNICODE` предписывает не преобразовывать многобайтовые символы (в том числе буквы кириллицы) в их коды.

Если на уровне модели, чьи объекты кодируются в формате JSON, реализуется немедленная выборка связанных записей первичных моделей (подробности — в *разд. 16.1*), эти связанные записи будут непосредственно включены в состав объектов модели. Пример:

```
// В модели Rubric объявлена немедленная загрузка записей первичной
// модели, связанных с текущей моделью связью parent
class Rubric extends Model {
    . . .
    protected $with = ['parent'];
    . . .
}
. . .
// При запросе по пути /api/rubrics/11/ будет выдан следующий ответ
// (связанная запись первичной модели включена в состав выдаваемой
// записи):
{
    "id": 11, "name": "Бытовая", "parent_id": "10",
    "created_at": "2022-05-16T16:13:08.000000Z",
    "updated_at": "2022-05-16T16:24:59.000000Z",
    "parent": {
        "id": 10, "name": "Техника", "parent_id": null,
        "created_at": "2022-05-15T10:15:34.000000Z",
        "updated_at": "2022-05-15T12:20:02.000000Z","parent": null
    }
}
```

- вернуть из действия контроллера результат вызова метода `toArray()`. Он полностью аналогичен методу `toJson()`, но не позволяет указать параметры кодирования. Примеры:

```
return Rubric::all()->toArray();

return $rubric->toArray();
```

Также можно использовать полностью аналогичный метод `jsonSerialize()`;

- если не требуется извлекать связанные записи — использовать метод `attributesToArray()`:

```
class Rubric extends Model {
    . . .
    protected $with = ['parent'];
    . . .
}
. . .
public function show(Rubric $rubric) {
    return $rubric->attributesToArray();
}
. . .
// При запросе по пути /api/rubrics/11/ будет выдан следующий ответ
// (связанная запись первичной модели в составе выдаваемой записи
// отсутствует):
{
    "id": 11, "name": "Бытовая", "parent_id": "10",
    "created_at": "2022-05-16T16:13:08.000000Z",
    "updated_at": "2020-05-16T16:24:59.000000Z"
}
```

- вернуть из действия контроллера результат приведения объекта модели или коллекции к строковому типу:

```
public function index() {
    return (string) Rubric::all();
}
```

- вернуть из действия контроллера непосредственно объект модели или коллекцию:

```
public function show(Rubric $rubric) {
    return $rubric;
}
```

Если требуется отправить клиенту закодированные в формат JSON произвольные данные, нужно использовать инструменты, описанные в *разд. 9.5.2.3*.

30.1.2. Задание структуры генерируемых JSON-объектов

По умолчанию при преобразовании записи модели в JSON-объект последний будет содержать все поля, присутствующие в кодируемой записи. Однако можно ограничить состав полей, включаемых в JSON-объект:

- на уровне модели — объявив в ее классе одно из следующих защищенных свойств:
  - `hidden` — хранит массив имен полей, значения которых *не* должны включаться в выдаваемые JSON-объекты (все остальные поля будут включены в них):

```
class Rubric extends Model {
    . . .
    protected $hidden = ['created_at', 'updated_at'];
    . . .
}
```

Также можно скрыть связи, добавив в массив из свойства `hidden` имена методов, создающих эти связи:

```
protected $hidden = ['created_at', 'updated_at', 'bbs'];
```

- `visible` — хранит массив имен полей, значения которых должны включаться в выдаваемые JSON-объекты (все остальные поля *не* будут включены в них):

```
class Rubric extends Model {
    . . .
    protected $visible = ['id', 'name', 'parent_id'];
    . . .
}
```

- на уровне текущего запроса — предварительно вызвав у объекта модели или коллекции записей один из следующих методов:
  - `makeHidden()` — исключает из выдаваемых JSON-объектов поля с заданными *именами*. Форматы вызова:

```
makeHidden(<имя поля 1>, <имя поля 2>, ... <имя поля n>)
makeHidden(<массив имен полей>)
```

Пример:

```
return Rubric::all()->makeHidden('created_at', 'updated_at')
                    ->toJson();
```

  - `makeHiddenIf(<условие>, <массив имен полей>)` — исключает из выдаваемых JSON-объектов поля с заданными *именами*, если указанное *условие* при вычислении выдает значение `true`:

```
return Rubric::all()->makeHiddenIf($excludeTimestamps,
                                   ['created_at', 'updated_at'])
                    ->toJson();
```

  - `makeVisible()` — включает в состав выдаваемых JSON-объектов поля с заданными *именами*, ранее помеченные как исключаемые. Форматы вызова такие же, как и у метода `makeHidden()`. Пример:

```
return Rubric::all()->makeVisible('created_at')->toJson();
```

  - `makeVisibleIf()` — включает в состав выдаваемых JSON-объектов поля с заданными *именами*, ранее помеченные как исключаемые, если указанное *условие* при вычислении выдает значение `true`. Формат вызова такой же, как и у метода `makeHiddenIf()`.

Также можно добавить в состав выдаваемых JSON-объектов произвольные значения. Для этого достаточно:

1. Для каждого из добавляемых значений — создать в классе модели виртуальное поле с произвольным именем. Пример такого виртуального поля можно увидеть в *разд. 5.6.1*.
2. Объявить в классе модели защищенное свойство `appends` и присвоить ему массив с именами созданных виртуальных полей, записанными в стиле snake_case:

```
class Rubric extends Model {
    . . .
    protected $appends = ['full_name'];
    . . .
}
```

После этого каждый выдаваемый JSON-объект будет включать значения указанных таким образом виртуальных полей.

Если присутствие каких-либо виртуальных полей во всех объектах модели не требуется, следует убрать эти поля из массива, хранящегося в свойстве `appends`, а при извлечении записей вызвать у объекта модели или коллекции один из следующих методов:

- `append()` — добавляет в выдаваемый JSON-объект виртуальные поля с указанными *именами*. Форматы вызова:

  ```
  append(<имя поля 1>, <имя поля 2>, ... <имя поля n>)
  append(<массив имен полей>)
  ```

  Пример:

  ```
  return $rubric->append('full_name')->toJson();
  ```

  Метод `append()` можно вызывать произвольное количество раз, при этом указываемые в его вызовах поля будут добавляться в состав включаемых в JSON-объекты:

  ```
  return $rubric->append('full_name')->append('bb_count')
                ->append('name_uppercased')->toJson();
  ```

- `setAppends(<массив имен полей>)` — указывает включать виртуальные поля с присутствующими в *массиве* именами в JSON-объекты, отменяя перечень выводимых полей, заданный предыдущими вызовами методов `append()` и `setAppends()`.

Значения дат и временны́х отметок при преобразовании в формат JSON будут сериализоваться в формат, заданный в свойстве `casts` класса модели (см. *разд. 5.3.3*):

```
class Rubric extends Model {
    protected $casts = [
        'created_at' => 'datetime:Y-m-d'
        'updated_at' => 'datetime:d-m-Y'
    ];
    . . .
}
```

Если все значения дат и временны́х отметок должны сериализоваться в одинаковом формате, можно реализовать необходимое преобразование в защищенном методе `serializeDate()` модели. С единственным параметром он должен принимать значение временно́й отметки типа `DateTimeInterface`, встроенного в PHP, и возвращать это значение в строковом виде, сериализованное в необходимом формате. Пример:

```
class Rubric extends Model {
    . . .
    protected function serializeDate(\DateTimeInterface $date) {
        return $date->format('Y-m-d');
    }
}
```

30.2. Бэкенды: преобразование данных — ресурсы и ресурсные коллекции

Если для преобразования моделей в формат JSON возможностей базовых инструментов окажется недостаточно, следует использовать ресурсы и ресурсные коллекции.

30.2.1. Ресурсы

Ресурс кодирует в JSON отдельный объект модели. Он позволяет задавать у свойств выдаваемых JSON-объектов произвольные имена, создавать свойства, хранящие связанные записи и коллекции связанных записей, и др. Ресурсы оформляются в виде классов.

30.2.1.1. Написание ресурсов

Новый класс ресурса создается подачей команды:

```
php artisan make:resource <имя класса ресурса> [--collection]
```

По действующим соглашениям имя класса создаваемого ресурса должно завершаться словом `Resource`.

Если указать ключ `--collection`, будет создана ресурсная коллекция. Ресурсная коллекция также будет создана, если указанное имя класса ресурса заканчивается словом `Collection`. Разговор о ресурсных коллекциях пойдет позже.

Новый класс ресурса объявляется в пространстве имен `App\Http\Resources` (соответствующая папка создается автоматически) и делается производным от класса `Illuminate\Http\Resources\Json\JsonResource`. Он содержит метод `toArray()`, который должен возвращать ассоциативный массив с данными, из которого будет создан JSON-объект. В теле этого метода можно получить доступ к полям объекта модели, представляемого ресурсом, обращаясь к одноименным свойствам ресурса. Изначально этот метод возвращает объект текущего запроса, преобразованный в массив вызовом того же метода базового класса.

В листинге 30.1 показан код ресурса `App\Http\Resources\RubricResource`, который преобразует в формат JSON отдельную рубрику, причем преобразованию подвергаются лишь поля `id`, `name` и `parent_id`.

Листинг 30.1. Код класса ресурса `App\Http\Resources\RubricResource`

```
namespace App\Http\Resources;
use Illuminate\Http\Resources\Json\JsonResource;
```

```
class RubricResource extends JsonResource {
    public function toArray($request) {
        return ['id' => $this->id, 'name' => $this->name,
                                   'parent_id' => $this->parent_id];
    }
}
```

Обратим внимание, как в теле метода `toArray()` осуществляется доступ к полям объекта модели, представляемого текущим ресурсом, — через одноименные свойства самого объекта ресурса.

Чтобы преобразовать объект модели в формат JSON с применением ресурса, следует создать объект ресурса, передав конструктору в качестве параметра преобразуемый объект модели. Полученный объект ресурса можно просто вернуть из действия контроллера в качестве результата. Пример:

```
use App\Http\Resources\RubricResource;
class ApiRubricController extends Controller {
    public function show(Rubric $rubric) {
        return new RubricResource($rubric);
    }
    . . .
}
```

JSON-объект, выдаваемый ресурсом, содержит единственное свойство `data`, хранящее запись модели:

```
{
    "data": {
        "id": 11, "name": "Бытовая", "parent_id": 10
    }
}
```

30.2.1.2. Задание структуры JSON-объектов, генерируемых ресурсами

В результирующие JSON-объекты можно заносить:

- поля объекта модели, хранящегося в текущем объекте ресурса (пример см. в листинге 30.1). Свойствам, хранящим значения полей, можно дать произвольные имена:

  ```
  public function toArray($request) {
      return ['rubric_key' => $this->id, 'rubric_title' => $this->name];
  }
  ```

- произвольные значения, в том числе получаемые в результате вычислений:

  ```
  return [ ... , 'bbs_count' => $this->bbs()->count()];
  ```

  Можно добавить в JSON-объект заданное *значение*, если оно отлично от `null`. Для этого следует использовать поддерживаемый ресурсом метод `whenNotNull(<значение>)`. Пример помещения в JSON-объект ключа рубрики первого уровня лишь в том случае, если он не равен `null` (то есть если текущая рубрика — второго уровня):

```
return [
    . . .
    'parent_id' => $this->whenNotNull($this->parent_id),
]
```

Также можно поместить заданное *значение* в JSON-объект только в том случае, если указанное *условие* при вычислении дает `true`. Для этого достаточно использовать метод `when()`, поддерживаемый ресурсом:

```
when(<условие>, <значение>|<анонимная функция>)
```

Пример добавления в JSON-объект количества связанных с рубрикой объявлений, только если рубрика содержит объявления:

```
return [
    . . .
    'bbs_count' => $this->when($this->bbs()->exists(),
                               $this->bbs()->count()),
];
```

Вместо *значения* можно указать не принимающую параметров *анонимную функцию* — тогда в массив будет помещен возвращенный ею результат:

```
return [
    . . .
    'bbs_count' => $this->when($this->bbs()->exists(), function () {
        return $this->bbs()->count();
    })
];
```

Если при истинности заданного *условия* нужно добавить в JSON-объект сразу несколько значений, содержащихся в указанном *массиве*, следует использовать метод `mergeWhen()`, поддерживаемый ресурсом:

```
mergeWhen(<условие>, <массив с добавляемыми значениями>)
```

Пример:

```
return [
    . . .
    $this->mergeWhen($this->bbs()->exists(), [
        'has_bbs' => true,
        'bbs_count' => $this->bbs()->count()
    ])
];
```

- единичные связанные записи — в виде объектов соответствующих ресурсов:

```
return [
    . . .
    'parent' => new RubricResource($this->parent),
];
```

Подобного рода ресурсы, являющиеся частью других ресурсов, носят название *вложенных*;

- коллекции связанных записей — в виде объектов ресурсных коллекций (будут описаны далее):

```
return [
    . . .
    'bbs' => new BbCollection($this->bbs),
];
```

Такие ресурсные коллекции носят название *вложенных*;

- значения из полей записи связующей таблицы (о связующих таблицах и связях «многие-со-многими» рассказывалось в *разд. 5.4.4*).

Такого рода значения могут быть получены лишь в том случае, когда Laravel знает, с какой записью второй из связываемых моделей связана текущая запись, и поэтому может найти запись связующей таблицы, которая связывает записи обеих моделей. В нашем случае такое может случиться, например, если извлекаются сведения о какой-либо конкретной машине, содержащие список входящих в ее состав деталей в качестве вложенной ресурсной коллекции.

Если же извлекаются сведения о конкретной детали без привязки к какой-либо машине, то фреймворку не удастся найти в связующей таблице соответствующую запись и поэтому он не сможет получить значение ее поля. В таком случае рекомендуется подстраховаться, реализуя загрузку значения поля из связующей таблицы только в том случае, если удается определить нужную запись связующей таблицы. Для этого используются следующие два метода, поддерживаемые объектом ресурса:

- `whenPivotLoaded()` — добавляет в JSON-объект значение поля заданной *связующей таблицы* только в том случае, если это значение удастся получить:

```
whenPivotLoaded(<связующая таблица>, <анонимная функция>)
```

В качестве *связующей таблицы* следует указать либо ее имя, либо «пустой» объект связующей модели (если таковая существует).

Само значение поля связующей таблицы должно возвращаться не принимающей параметров *анонимной функцией*, указанной в вызове метода. Пример:

```
return [
    . . .
    'count' => $this->whenPivotLoaded('machine_spare',
        function () {
            return $this->pivot->cnt;
        }
    ),
];
```

Метод `whenPivotLoaded()` использует для доступа к записи связующей таблицы свойство модели с именем по умолчанию: `pivot`;

- `whenPivotLoadedAs()` — аналогичен `whenPivotLoaded()`, только позволяет указать *имя свойства* модели для доступа к записи связующей таблицы, если это поле имеет имя, отличное от используемого по умолчанию:

```
whenPivotLoadedAs(<имя свойства с записью связующей таблицы>,
                  <связующая таблица>, <анонимная функция>)
```

Пример:

```
return [
    . . .
    'count' => $this->whenPivotLoadedAs('connector',
                                        new MachineSpare,
        function () {
            return $this->connector->cnt;
        }
    )
];
```

- произвольные служебные метаданные:

```
return [
    . . .
    'powered_by' => 'Laravel',
];
```

Если метаданные следует включить в массив только в том случае, когда текущий ресурс *не* является вложенным, надо объявить в классе ресурса общедоступный метод `with()`. В качестве параметра он должен принимать объект запроса и возвращать ассоциативный массив с включаемыми метаданными. Пример:

```
class RubricResource extends JsonResource {
    . . .
    public function with($request) {
        return ['powered_by' => 'Laravel'];
    }
}
```

30.2.1.3. Дополнительные параметры ресурсов

Ранее говорилось, что ресурс выдает JSON-объект со свойством `data`, которое и хранит запись модели. Этому свойству можно дать другое имя, присвоив его статическому общедоступному свойству `wrap` класса ресурса. Пример:

```
class RubricResource extends JsonResource {
    public static $wrap = 'rubric';
    . . .
}
. . .
// Результат
{
    "rubric": {
        "id": 11, "name": "Бытовая", "parent_id": 10
    }
}
```

Новое *имя* этого *свойства* также можно указать в вызове статического метода `wrap(<имя свойства>)` у нужного класса ресурса. Вызов этого метода следует поместить в теле метода `boot()` какого-либо провайдера (например, `AppServiceProvider`). Пример:

```
use App\Http\Resources\RubricResource;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        RubricResource::wrap('rubric');
    }
}
```

Можно вообще убрать из JSON-объекта свойство `data` — тогда он непосредственно будет хранить запись. Для этого следует в теле метода `boot()` какого-либо провайдера поместить вызов статического метода `withoutWrapping()` у нужного класса ресурса. Пример:

```
use App\Http\Resources\MachineResource;
class AppServiceProvider extends ServiceProvider {
    . . .
    public function boot() {
        . . .
        MachineResource::withoutWrapping();
    }
}
. . .
// Результат
{
    "id": 11, "name": "Бытовая", "parent_id": 10
}
```

Наконец, можно указать заголовки, которые будут добавлены в серверный ответ с отправляемым клиенту JSON-объектом. Для этого следует объявить в классе ресурса общедоступный метод `withResponse()`, принимающий в качестве параметров объекты запроса и ответа. Нужные заголовки задаются в теле этого метода с помощью метода `header()` ответа (см. *разд. 9.5.3*). Пример:

```
class RubricResource extends JsonResource {
    . . .
    public function withResponse($request, $response) {
        $response->header('X-Data-Kind', 'rubric');
    }
}
```

30.2.1.4. Использование ресурсов

Чтобы преобразовать какую-либо запись модели в формат JSON, достаточно создать объект ресурса, передав ему преобразуемую запись в качестве параметра. Готовый объект ресурса можно просто вернуть из действия контроллера в качестве результата (пример можно увидеть в *разд. 30.2.1.1*).

Все объекты ресурсов поддерживают метод `additional(<`*массив*`>)`, добавляющий в результирующий JSON-объект новые свойства. В заданном ассоциативном *массиве* ключи элементов зададут имена свойств, а значения элементов — значения этих свойств. Пример:

```
public function show(Rubric $rubric) {
    return (new RubricResource($rubric))
                ->additional(['powered_by' => "Laravel']);
}
```

Метод `response()`, также поддерживаемый всеми ресурсами, возвращает сформированный на основе текущего ресурса серверный ответ. Это может пригодиться, если требуется добавить в ответ какие-либо заголовки. Пример:

```
public function show(Rubric $rubric) {
    return (new RubricResource($rubric))->response()
                                        ->header('X-Data-Kind', 'rubric');
}
```

30.2.2. Ресурсные коллекции

Ресурсная коллекция преобразует в формат JSON коллекцию записей, используя для кодирования отдельных записей связанный с ней ресурс и при необходимости реализуя пагинацию.

30.2.2.1. Быстрое JSON-кодирование коллекции записей

Если нужно кодировать коллекцию записей какой-либо модели в JSON без изменений, объявлять для этого класс ресурсной коллекции необязательно — можно использовать класс ресурса, ранее объявленного для кодирования записей этой модели. У класса ресурса нужно вызвать статический метод `collection(<`*коллекция записей*`>)`. Пример:

```
class ApiRubricController extends Controller {
    public function index() {
        return RubricResource::collection(Rubric::all());
    }
    . . .
}
```

30.2.2.2. Написание ресурсных коллекций

Новый класс ресурсной коллекции создается командой `make:resource` утилиты artisan (см. *разд. 30.2.1.1*), для чего достаточно либо завершить указываемое имя класса словом `Collection`, либо задать ключ `--collection`.

Класс ресурсной коллекции объявляется в том же пространстве имен `App\Http\Resources` и делается производным от класса `Illuminate\Http\Resources\Json\ResourceCollection`. Он содержит метод `toArray()`, который должен возвращать ассоциативный массив с данными, подлежащими преобразованию в формат JSON.

В листинге 30.2 показан код ресурсной коллекции `App\Http\Resources\RubricCollection`, которая преобразует в формат JSON коллекцию рубрик.

Листинг 30.2. Код класса ресурсной коллекции `App\Http\Resources\RubricCollection`

```
namespace App\Http\Resources;
use Illuminate\Http\Resources\Json\ResourceCollection;
```

```
class RubricCollection extends ResourceCollection {
    public function toArray($request) {
        return ['data' => $this->collection];
    }
}
```

Чтобы преобразовать коллекцию записей в формат JSON, следует создать объект ресурсной коллекции, передав конструктору в качестве параметра преобразуемую коллекцию записей. Пример:

```
use App\Http\Resources\RubricCollection;
class ApiRubricController extends Controller {
    public function index() {
        return new RubricCollection(Rubric::all());
    }
    . . .
}
. . .
// Результат
{
    "data": [
        { "id": 1, "name": "Здания", "parent_id": null },
        { "id": 2, "name": "Дома", "parent_id": 1 },
        { "id": 3, "name": "Гаражи", "parent_id": 1 },
        { "id": 4, "name": "Транспорт", "parent_id": null },
        . . .
    ]
}
```

Ресурсные коллекции могут создавать в генерируемых JSON-объектах свойства тех же типов, что и ресурсы (см. *разд. 30.2.1.2*), за исключением разве что значений полей связующей таблицы (по вполне понятным причинам).

Для преобразования в формат JSON отдельных записей ресурсная коллекция использует класс ресурса, чье имя схоже с именем класса самой ресурсной коллекции без суффикса `Collection` с добавленным суффиксом `Resource` (так, ресурсная коллекция `RubricCollection` будет использовать ресурс `RubricResource`). Можно указать для этого другой класс ресурса, записав путь к нему в общедоступное свойство `collect` класса ресурсной коллекции. Пример:

```
class RubricCollection extends ResourceCollection {
    public $collect = 'App\Http\Resources\RubRes';
    . . .
}
```

Элементу возвращаемого методом `toArray()` массива, хранящему коллекцию записей, желательно давать имя `data`. Если дать ему другое имя, в результирующем JSON-объекте в свойстве `data` будет создан вложенный объект со свойством, имя которого совпадает с именем упомянутого ранее элемента, и уже это свойство будет хранить массив записей. Пример:

```
class RubricCollection extends ResourceCollection {
    public function toArray($request) {
        return ['rubrics' => $this->collection];
    }
}
. . .
// Результат
{
    "data":{
        rubrics:[
            { "id": 1, "name": "Здания", "parent_id": null },
            { "id": 2, "name": "Дома", "parent_id": 1 },
            { "id": 3, "name": "Гаражи", "parent_id": 1 ,
            { "id": 4, "name": "Транспорт", "parent_id": null },
            . . .
        ]
    }
}
```

Можно указать ресурсной коллекции создавать для хранения записей не массив, а вложенный объект. Этот объект будет содержать свойства с именами, совпадающими со значениями какого-либо уникального поля записей (обычно ключа), и хранить эти свойства будут соответствующие записи. Для этого достаточно:

1. Добавить в класс ресурсной коллекции общедоступное свойство `preserveKeys` со значением `true`:

   ```
   class RubricCollection extends ResourceCollection {
       public $preserveKeys = true;
       . . .
   }
   ```

2. При создании ресурсной коллекции — вызвать у передаваемой конструктору ее класса коллекции записей метод `keyBy()` (см. *разд. 15.1.11*), передав ему в качестве параметра имя нужного поля:

   ```
   return new RubricCollection(Rubric::all()->keyBy('id'));
   ```

Результат:

```
{
    "data": {
        "1": { "id": 1, "name": "Здания", "parent_id": null },
        "2": { "id": 2, "name": "Дома", "parent_id": 1 },
        "3": { "id": 3, "name": "Гаражи", "parent_id": 1 },
        "4": { "id": 4, "name": "Транспорт", "parent_id": null },
        . . .
    }
}
```

30.2.2.3. Пагинация в ресурсных коллекциях

Если вместо коллекции записей передать конструктору класса ресурсной коллекции объект пагинатора (см. *главу 12*), ресурсная коллекция сгенерирует JSON-нотацию не всей коллекции, а текущей части этого пагинатора. Пример:

```
public function index() {
    return new RubricCollection(Rubric::paginate(3));
}
```

В этом случае полученный JSON-объект будет содержать следующие свойства:

- `data` — массив с записями, входящими в выбранную часть;
- `links` — полные интернет-адреса других частей пагинатора. Содержит вложенный JSON-объект со свойствами:
  - `first` — интернет-адрес первой части;
  - `last` — интернет-адрес последней части;
  - `prev` — интернет-адрес предыдущей части или `null`, если предыдущей части нет;
  - `next` — интернет-адрес следующей части или `null`, если следующей части нет.

  Все эти интернет-адреса создаются на основе текущего интернет-адреса, по которому был выполнен запрос;
- `meta` — сведения о части и самом пагинаторе в виде вложенного JSON-объекта со свойствами:
  - `current_page` — порядковый номер текущей части, начиная с 1;
  - `last_page` — порядковый номер последней части пагинатора, начиная с 1;
  - `per_page` — предельное количество записей, входящих в часть;
  - `from` — порядковый номер первой записи, входящей в текущую часть, начиная с 1;
  - `to` — порядковый номер последней записи, входящей в текущую часть, начиная с 1;
  - `total` — общее количество записей в коллекции;
  - `path` — текущий интернет-адрес.

Класс ресурсной коллекции поддерживает два полезных метода:

- `withQuery(<`*массив GET-параметров*`>)` — указывает включить в интернет-адреса частей пагинатора GET-параметры, приведенные в заданном ассоциативном *массиве*. Ключи его элементов зададут имена GET-параметров, а значения элементов — их значения. Пример:

  ```
  public function index() {
      return new RubricCollection(Rubric::paginate(3))
              ->withQuery(['search' => 'дом']);
  }
  ```

- `preserveQuery()` — указывает включить в интернет-адреса частей пагинатора все GET-параметры, что присутствуют в текущем интернет-адресе.

30.3. Бэкенды: обработка данных

30.3.1. Выдача записей

Проще всего реализовать в бэкенде выдачу коллекций записей. Вот пример маршрута и действия контроллера, выдающего фронтенду перечень рубрик:

```
// API-маршруты записываются в модуле routes\api.php
Route::get('/rubrics', [ApiRubricController::class, 'index']);
. . .
use App\Http\Resources\RubricCollection;
class ApiRubricController extends Controller {
    public function index() {
        return new RubricCollection(Rubric::all());
    }
    . . .
}
```

При выборке отдельной записи удобно использовать внедрение модели. Если требуемую запись в базе данных найти не удалось, фреймворк самостоятельно сгенерирует и вернет фронтенду ответ в формате JSON с кодом статуса 404. Пример:

```
Route::get('/rubrics/{rubric}', [ApiRubricController::class, 'show']);
. . .
use App\Http\Resources\RubricResource;
class ApiRubricController extends Controller {
    . . .
    public function show(Rubric $rubric) {
        return new RubricResource($rubric);
    }
    . . .
}
```

30.3.2. Добавление, правка и удаление записей

При программировании добавления, правки и удаления записей также можно положиться на высокоуровневые механизмы Laravel — в частности, на внедрение моделей и валидаторы.

Все валидаторы проверяют, присутствует ли в клиентском запросе заголовок `Accept` со значением `application/json`, что является сигналом того, что клиент, приславший запрос, желает получить ответ в формате JSON. В этом случае, если данные не прошли валидацию, валидатор сам отправит клиенту ответ с кодом статуса 422 (данные невозможно обработать) и JSON-объектом, содержащим сообщения об ошибках. Этот JSON-объект включает следующие свойства:

- `message` — строка с сообщением (например: «The given data was invalid.», предоставленные данные некорректны);
- `errors` — вложенный объект с сообщениями об ошибках. Содержит свойства с именами, идентичными наименованиям элементов управления. Значениями этих свойств являются массивы со строковыми сообщениями об ошибках.

При добавлении записи, если введенные пользователем данные пройдут валидацию, следует добавить в базу данных новую запись, преобразовать ее в формат JSON и вернуть в составе ответа с кодом статуса 201 (данные успешно добавлены), чтобы фронтенд смог вывести новую запись на экран. Если запись не предназначена для немедленного вывода, можно вернуть «пустой» ответ, также с кодом 201. Пример:

```
Route::post('/rubrics', [ApiRubricController::class, 'store']);
. . .
class ApiRubricController extends Controller {
    . . .
    public function store(Request $request) {
        . . .
        // Предполагается, что правила валидации хранятся в переменной
        // validationRules
        $validated = $request->validate($validationRules);
        $rubric = Rubric::create($validated);
        return response()->json(new RubricResource($rubric), 201);
    }
}
```

При правке записи все происходит аналогично, за тем исключением, что в случае успешной правки записи нужно вернуть ответ с кодом статуса 200 (поскольку этот код статуса подставляется в ответ по умолчанию, в вызове метола `json()` его указывать необязательно):

```
Route::patch('/rubrics/{rubric}',
             [ApiRubricController::class, 'update']);
. . .
class ApiRubricController extends Controller {
    . . .
    public function update(Request $request, Rubric $rubric) {
        . . .
        $validated = $request->validate($validationRules);
        $rubric->update($validated);
        return response()->json(new RubricResource($rubric));
        // А можно сделать еще проще:
        // return new RubricResource($rubric);
    }
    . . .
}
```

При удалении записи в случае успеха нужно вернуть «пустой» ответ с кодом статуса 204 (данные отсутствуют):

```
Route::delete('/rubrics/{rubric}',
              [ApiRubricController::class, 'destroy']);
. . .
class ApiRubricController extends Controller {
    . . .
    public function destroy(Rubric $rubric) {
        $rubric->delete();
```

```
        return response()->noContent(204);
    }
}
```

30.3.3. Совмещенная обработка данных

Разработчики Laravel предлагают создателям сайтов разделять функциональность традиционного сайта и веб-службы. Они считают, что, например, веб-страницу со списком рубрик должно выводить одно действие контроллера, а JSON-объект со списком рубрик — другое. Для записей маршрутов, ведущих на эти действия, даже предусмотрены два разных модуля: routes\web.php и routes\api.php.

Однако можно без проблем совместить вывод страницы и генерирование JSON-объекта в одном действии. Для определения, какой формат данных требует клиент: HTML или JSON — можно использовать метод `expectsJson()` запроса (см. *разд. 9.3.3*). Пример:

```
// Модуль routes\web.php
Route::get('rubrics', [RubricController::class, 'index']);
. . .
class RubricController extends Controller {
    public function index(Request $request) {
        if ($request->expectsJson()) {
            return new RubricCollection(Rubric::all());
        } else {
            return view('rubrics.index', ['rubrics' => Rubric::all()]);
        }
    }
    . . .
}
```

30.4. Бэкенды: разграничение доступа

Удобнее всего в бэкендах применять *вход по жетону*, или *жетонную аутентификацию*. Суть ее состоит в том, что каждый зарегистрированный пользователь идентифицируется по уникальному электронному жетону, назначаемому ему либо непосредственно при регистрации, либо позже, при выполнении первой процедуры входа. Такой жетон сохраняется в составе сведений о пользователе.

При выполнении процедуры входа фронтенд отправляет бэкенду адрес электронной почты и пароль пользователя, а бэкенд в ответ пересылает фронтенду жетон пользователя (генерируя его, если он не был сгенерирован ранее, при регистрации). Процедура получения жетона называется *подключением к бэкенду*.

При попытке доступа к закрытым данным фронтенд отправляет бэкенду в составе запроса (в GET-, POST-параметре или заголовке) полученный ранее жетон. Бэкенд проверит, присутствует ли этот жетон в списке пользователей, и в случае успеха допустит пользователя до закрытых данных.

Для защиты маршрутов от неавторизованного доступа в Laravel применяется посредник `App\Http\Middleware\Authenticate`, имеющий краткое обозначение `auth`. У этого посредника следует указать страж `api`, который будет самостоятельно извлекать из

запроса переданный фронтендом жетон и проверять его на наличие в списке пользователей. К сожалению, этот страж в настройках проекта не указан, но его нетрудно добавить туда.

Чтобы реализовать вход по жетону, который представляет собой строку из случайных символов и генерируется в момент запроса его фронтендом, нужно:

1. Добавить в модуль настроек config\auth.php конфигурацию стража `api` (подробное описание используемых для этого настроек приведено в *разд. 13.1*):

```
'guards' => [
    . . .
    'api' => [
        'driver' => 'token',
        'provider' => 'users',
        'hash' => true,
    ],
],
```

   Обязательно следует дать настройке стража `hash` значение `true`, указав тем самым фреймворку, что жетон будет храниться в списке пользователей хешированным с использованием алгоритма SHA256, — это повысит безопасность.

   Далее подразумевается, что поле списка пользователей, в котором будет храниться жетон, и GET- или POST-параметр, через который он будет пересылаться, будут иметь имена по умолчанию: `api_token`. Если это не так, следует указать другие имена в настройках стража `storage_key` и `input_key`, а также внести соответствующие изменения в приведенный далее код.

2. Добавить в таблицу списка пользователей поле `api_token`, в котором будет храниться электронный жетон. Это поле должно быть строковым, иметь длину 80 символов (именно такую длину имеет хеш, сгенерированный алгоритмом SHA256), уникальный индекс и не являться обязательным для заполнения.

   Добавить такое поле можно, написав миграцию со следующим кодом (предполагается, что таблица списка пользователей имеет имя по умолчанию `users`):

```
class UpdateInUsersTable extends Migration {
    public function up() {
        Schema::table('users', function (Blueprint $table) {
            $table->string('api_token', 80)->unique()->nullable()
                                            ->default(null);
        });
    }
    . . .
}
```

   Не забудьте применить эту миграцию.

3. Создать в модуле routes\api.php маршрут, указывающий на действие контроллера, которое будет проверять полученные от фронтенда адрес электронной почты и пароль и выдавать жетон, а также написать само это действие.

   Действие следует сделать доступным только для гостей, привязав к указывающему на него маршруту посредник `guest`. Если пользователь с полученными адресом

электронной почты и паролем не найден, следует вернуть «пустой» ответ с кодом 401 (пользователь не представился). Пример:

```
Route::post('/login', [ApiLoginController::class, 'login'])
     ->middleware('guest');
. . .
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;
use App\Models\User;
class ApiLoginController extends Controller {
    public function login(Request $request) {
        $email = $request->email;
        $password = $request->password;
        $user = User::firstWhere('email', $email);
        if ($user && Hash::check($password, $user->password)) {
            if (!$user->api_token) {
                $token = Str::random(80);
                User::where('id', $user->id)
                    ->update(['api_token' => hash('sha256', $token)]);
            }
            return response()->json(['api_token' => $api_token]);
        } else {
            return response()->noContent(401);
        }
    }
    . . .
}
```

Действие генерирует жетон, если он не был сгенерирован ранее, в виде строки со случайным набором символов, сохраняет его хеш, вычисленный по алгоритму SHA256, в поле `api_token` таблицы списка пользователей и высылает фронтенду в составе ответа.

Отметим, что для сохранения хеша жетона в списке пользователей применяется метод `update()` модели (о нем и вообще о массовой правке записей рассказывалось в *разд. 6.2.2*). Это сделано для того, чтобы после сохранения очередного жетона в записи в нее не заносилась новая отметка правки и, соответственно, запись не помечалась как исправленная (что в некоторых случаях может оказаться неприемлемым).

Получив жетон, фронтенд сохранит его и впоследствии использует для получения закрытых данных от бэкенда.

4. Связать маршруты, ведущие на закрытые данные, с посредником `auth`, указав у него страж `api`:

```
Route::post('/rubrics', [ApiRubricController::class, 'store'])
     ->middleware('auth:api');
```

При выполнении процедуры выхода хеш жетона, сохраненный в списке пользователей, будет удален. При последующем подключении произойдет генерация нового жетона.

5. Создать маршрут, указывающий на действие, которое реализует выход, и само это действие.

   Действие нужно сделать доступным только для пользователей, выполнивших подключение к бэкенду, связав маршрут с посредником `auth` и стражем `api`. После успешного выхода обычно возвращают «пустой» ответ с кодом состояния 204 (данные отсутствуют — в нашем случае это значит, что выход прошел успешно). Пример:

```
Route::get('/logout', [ApiLoginController::class, 'logout'])
     ->middleware('auth:api');
. . .
class ApiLoginController extends Controller {
    . . .
    public function logout(Request $request) {
        $user = $request->user();
        User::where('id', $user->id)->update(['api_token' => null]);
        return response()->noContent(204);
    }
}
```

Возможны варианты реализации разграничения доступа. Так, можно генерировать жетоны по пользовательскому запросу, отправленному со специальной веб-страницы, — тогда жетоны получат лишь те пользователи, которым они действительно нужны. Жетоны можно также генерировать на базе сведений о пользователе — например, в виде хеша, вычисленного на основе имени пользователя или его адреса электронной почты.

30.5. Фронтенды: взаимодействие с бэкендами

Для взаимодействия с бэкендами, написанными на Laravel, можно использовать любые клиентские JavaScript-библиотеки, равно как и инструменты самого JavaScript и HTML API, — в частности метод `fetch()` класса `Window`. Единственное, что необходимо сделать, — добавить в отправляемый запрос заголовок `Accept` со значением `application/json`. Вот пример загрузки с бэкенда списка рубрик:

```
async function loadRubrics() {
    const response = await fetch(
        'http://localhost:8000/api/rubrics',
        {
            headers: {
                'Accept': 'application/json'
            }
        }
    );
    const data = await response.json();
    // Обрабатываем полученный список рубрик
}
```

Если требуется получить данные, доступные лишь зарегистрированным пользователям, сначала следует выполнить вход. Пример кода, выполняющего вход и сохраняющего

полученный жетон в переменной `token` (это сделано для простоты — в реальных фронтендах желательно записать его в локальное или сессионное хранилище HTML API):

```
let token;
async function login() {
    const fd = new FormData();
    fd.append('email', 'editor@bboard.ru');
    fd.append('password', 'supereditor');
    const response = await fetch(
        'http://localhost:8000/api/login',
        {
            method: 'post',
            headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/x-www-form-urlencoded'
            },
            body: fd
        }
    );
    const data = await response.json();
    token = data.api_token;
}
```

Полученный электронный жетон можно использовать для доступа к закрытым ресурсам. Отправить его бэкенду можно тремя способами:

- в GET-параметре `api_token`:

```
const response = await fetch(
    'http://localhost:8000/api/rubrics?api_token=' + token,
    {
        headers: {
            'Accept': 'application/json'
        }
    }
);
```

- в POST-параметре `api_token`:

```
const fd = new FormData();
fd.append('name', 'Одежда');
fd.append('parent_id', '');
fd.append('api_token', token);
const response = await fetch(
    'http://localhost:8000/api/rubrics,
    {
        method: 'post',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/x-www-form-urlencoded'
        },
```

```
            body: fd
        }
    );
```

- в заголовке запроса `Authorization` в виде значения формата `Bearer <жетон>`:

```
const response = await fetch(
    'http://localhost:8000/api/rubrics,
    {
        method: 'post',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/x-www-form-urlencoded',
            'Authorization': 'Bearer ' + token
        },
        body: fd
    }
);
```

30.6. Фронтенды: использование React и Vue

Для написания фронтендов можно использовать популярные клиентские JavaScript-фреймворки React и Vue. Команда `ui`, добавляющаяся в утилиту artisan после установки библиотеки laravel/ui, поможет установить в составе проекта эти фреймворки, подготовить проект к их использованию и даже создать простейшие тестовые компоненты, которые можно задействовать для проверки работы React и Vue.

Подготовку проекта к использованию этих JavaScript-фреймворков выполняет следующая команда:

```
php artisan ui react|vue [--auth]
```

В результате команда `ui`:

- добавит в проект:
  - React-компонент resources\js\components\Example.js — если был выбран фреймворк React;
  - Vue-компонент resources\js\components\ExampleComponent.vue — если был выбран Vue.

  Оба компонента чрезвычайно просты и лишь выводят статичный текст;
- перезапишет:
  - модуль resources\js\app.js — его новая редакция включит необходимый импортирующий и инициализационный код;
  - файл webpack.mix.js — его новая редакция будет содержать конфигурацию пакета Laravel Mix (см. *разд. 17.5*), необходимую для успешной обработки файлов с кодом выбранного JavaScript-фреймворка;
- исправит — файл package.json, добавив в него все необходимые зависимости;

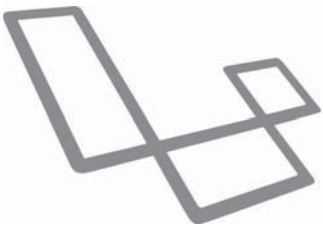
- удалит папку node_modules, если таковая присутствует, вместе с находящимися в ней библиотеками.

При указании командного ключа `--auth` в проект также будут добавлены контроллеры, реализующие разграничение доступа, и шаблоны необходимых страниц (они были описаны в *главе 13*).

После выполнения команды `ui` следует установить все необходимые Node-модули, набрав команду:

```
npm install
```

И можно начинать программирование фронтенда с применением выбранного ранее JavaScript-фреймворка.



ГЛАВА 31

Вещание

Обмен данными по протоколам HTTP и HTTPS может быть инициирован только клиентом. Следовательно, клиент, ожидающий отправки сервером каких-либо данных, вынужден регулярно опрашивать сервер, выясняя, готов ли он отправить данные. Это создает дополнительную нагрузку и на сервер, и на клиент.

В качестве альтернативы можно использовать подсистему *вещания*, встроенную в Laravel. Если серверу потребуется отправить данные клиенту, он оформит их в виде вещаемого события или оповещения и отправит клиенту по одному из предусмотренных разработчиком сайта *каналов вещания* с применением протокола WebSocket. Клиент, прослушивающий этот канал, тотчас получит отправленные сервером данные.

Каналы вещания могут быть как общедоступными, так и закрытыми. Для подключения к последним требуется выполнить вход на сайт.

31.1. Бэкенд: подготовка подсистемы вещания

Чтобы подготовить встроенную во фреймворк подсистему вещания к работе, нужно указать необходимые настройки и, возможно, установить дополнительные программы.

31.1.1. Настройка подсистемы вещания

Настройки подсистемы вещания хранятся в модуле config\broadcasting.php:

- `connections` — содержит ассоциативный массив служб, реализующих вещание. Ключи элементов массива задают имена служб, а значениями являются вложенные ассоциативные массивы с параметрами отдельных служб. Доступны следующие параметры:
  - `driver` — тип службы. Поддерживаются значения:
    - `pusher` — веб-служба каналов доставки данных Pusher Channels (**https://pusher.com/channels**). Требует установки дополнительной библиотеки набором команды:

      ```
      composer require pusher/pusher-php-server
      ```

Вместо службы Pusher Channel может быть использован бесплатный сервер вещания Laravel Websockets;

- `ably` — веб-служба каналов доставки Ably (**https://ably.io/**). Требует установки дополнительной библиотеки набором команды:

  ```
  composer require ably/ably-php
  ```

- `redis` — «связка» из нереляционной базы данных Redis и сокет-сервера laravel-echo-server, устанавливаемого отдельно.

  В текущей версии Laravel эта служба работает крайне неустойчиво, вследствие чего настоятельно не рекомендуется к применению. Не исключено, что разработчики со временем прекратят ее сопровождение;

- `log` — служба журналирования Laravel. Не выполняет отправку событий и оповещений, а лишь регистрирует их в журнале. Используется только при отладке;

- `null` — отключает подсистему вещания.

Следующие настройки используются только службой `pusher`:

- `key` — ключ пользователя службы. Значение берется из локальной настройки `PUSHER_APP_KEY`, которая хотя и присутствует в файле .env, но изначально «пуста»;
- `secret` — секретный ключ пользователя службы. Значение берется из локальной настройки `PUSHER_APP_SECRET`, которая хотя и присутствует в файле .env, но изначально «пуста»;
- `app_id` — идентификатор приложения службы. Значение берется из локальной настройки `PUSHER_APP_ID`, которая хотя и присутствует в файле .env, но изначально «пуста»;
- `options` — дополнительные параметры. Значение настройки представляет собой ассоциативный массив со следующими параметрами:
  - `cluster` — обозначение используемого кластера службы. Значение берется из локальной настройки `PUSHER_APP_CLUSTER`, имеющей изначальное значение `mt1`;
  - `useTLS` — если `false`, подключение к службе Pusher Channels будет выполняться по незащищенному протоколу, если `true` — по протоколу TLS (изначально: `true`);

  Следующие настройки используются для подключения к серверу, выступающему заменой службе Pusher Channels (например, Laravel Websockets):

  - `host` — интернет-адрес сервера;
  - `port` — номер TCP-порта сервера;
  - `scheme` — обозначение протокола, с помощью которого будет выполняться обмен данными с сервером: `'http'` или `'https'`.

Следующая настройка используется только службой `ably`:

- `key` — ключ пользователя службы. Значение берется из локальной настройки `ABLY_KEY`, изначально отсутствующей;

Следующая настройка используется только службой `redis`:

- `connection` — имя базы данных из приведенных в настройках `redis.connections` (см. *разд. 25.1.3*). По умолчанию: `default`.

Изначально созданы службы: `redis`, `pusher`, `ably`, `log` и `null`;

- `default` — используемая служба вещания. Значение берется из локальной настройки `BROADCAST_DRIVER`, имеющей изначальное значение `log`. По умолчанию: `null`.

31.1.2. Подготовка проекта к реализации вещания

Чтобы подготовить проект должным образом, необходимо выполнить следующие действия:

1. Настроить и подготовить к работе подсистему очередей (см. *главу 25*).

   Вещаемые события и оповещения сначала заносятся в очередь в виде отложенных заданий, а уже потом отправляются службе вещания — для повышения отзывчивости сайта.

2. Открыть модуль настроек config\app.php и раскомментировать провайдер `App\Providers\BroadcastServiceProvider`.

   Этот провайдер загружает список маршрутов каналов и создает веб-маршрут с шаблонным путем **/broadcasting/auth**, указывающий на действие встроенного во фреймворк контроллера, которое выполняет авторизацию клиентов для подключения к закрытым каналам и каналам присутствия.

31.1.3. Установка и настройка Laravel Websockets

Laravel Websockets — это полностью бесплатная программа сервера вещания, которая написана командой разработчиков Laravel и может быть использована в качестве замены службы Pusher Channels.

> ***Полная документация по Laravel Websockets…***
> …находится по интернет-адресу: **https://beyondco.de/docs/laravel-websockets**.

Для установки сервера вещания в составе текущего проекта надо набрать команду:

```
composer require beyondcode/laravel-websockets
```

Далее необходимо выполнить следующие действия:

- перенести в состав проекта миграцию, поставляемую в составе Laravel Websockets, набрав команду:

  ```
  php artisan vendor:publish ↳
  --provider="BeyondCode\LaravelWebSockets\WebSocketsServiceProvider" ↳
  --tag="migrations"
  ```

  Эта миграция создает в базе данных по умолчанию таблицу для хранения статистики работы сервера (которую, впрочем, вести необязательно);

- применить миграции;

- перенести в состав проекта модуль настроек сервера, для чего набрать команду:

```
php artisan vendor:publish ↳
--provider="BeyondCode\LaravelWebSockets\WebSocketsServiceProvider" ↳
--tag="config"
```

Один экземпляр сервера Laravel Websockets может работать с произвольным количеством служб вещания типа `pusher` (была описана в *разд. 31.1.1*), реализованных в составе сайта. Это может оказаться полезным при программировании сложных сайтов, рассылающих множество вещаемых событий, которые для удобства разработки распределены по нескольким службам вещания.

Для работы с каждой отдельной службой вещания в настройках Laravel Websockets создается отдельная точка входа, называемая приложением (app).

Настройки сервера вещания записываются в модуле config\websockets.php (здесь приведены наиболее важные, остальные описаны в документации по этой программе):

- `apps` — массив приложений (точек входа для работы со службой вещания). Каждый элемент этого массива представляет собой ассоциативный массив с собственно настройками отдельных приложений:
  - `app_id` — идентификатор приложения;
  - `name` — имя приложения;
  - `key` — ключ пользователя;
  - `secret` — секретный ключ пользователя.

  Значения этих четырех параметров могут быть выбраны произвольно;
  - `enable_client_messages` — если `true`, сервер будет поддерживать каналы присутствия, если `false` — не будет (по умолчанию: `false`);
  - `enable_statistics` — если `true`, сервер будет вести статистику работы, если `false` — не будет (по умолчанию: `true`).

  Можно создать произвольное количество приложений — каждое для работы с одной из служб вещания.

  Изначально создано одно приложение, которое получает значения идентификатора, имени, обычного и секретного ключей из локальных настроек `PUSHER_APP_ID`, `APP_NAME`, `PUSHER_APP_KEY` и `PUSHER_APP_SECRET` соответственно;
- `dashboard` — настройки подключения к серверу. Представляют собой ассоциативный массив собственно настроек:
  - `port` — номер TCP-порта, через который фронтенды будут подключаться к серверу. Значение берется из локальной настройки `LARAVEL_WEBSOCKETS_PORT`, изначально отсутствующей. По умолчанию: 6001;
- `allowed_origins` — массив хостов, с которых можно будет производить подключение к серверу вещания. Если массив «пуст», поддерживаются подключения с любого хоста. По умолчанию: «пустой» массив.

Настройка сервера вещания в простейшем случае будет заключаться лишь в указании произвольных значений идентификатора, обычного и секретного ключей пользователя. Их удобнее всего занести в локальные настройки `PUSHER_APP_ID`, `PUSHER_APP_KEY` и

PUSHER_APP_SECRET (поскольку из этих же настроек они будут загружаться и подсистемой вещания фреймворка). Пример:

```
PUSHER_APP_ID=12121
PUSHER_APP_KEY=12345
PUSHER_APP_SECRET=54321
```

Далее необходимо записать в массив из настройки broadcasting.stores.pusher.options параметры для подключения к серверу вещания:

```
'connections' => [
    'pusher' => [
        'driver' => 'pusher',
        . . .
        'options' => [
            'cluster' => env('PUSHER_APP_CLUSTER'),
            'host' => '127.0.0.1',
            'port' => 6001,
            'scheme' => 'http',
        ],
        . . .
    ],
    . . .
],
```

Запуск сервера вещания Laravel Websockets производится командой:

```
php artisan websockets:serve
```

В процессе работы сервер выводит непосредственно в командной строке довольно подробный журнал, включающий сведения о подключении к каналам вещания, отключении от них, авторизации перед подключением к закрытым каналам и каналам присутствия, вещаемых событиях и оповещениях, пересылаемых по каналам, а также возникающих ошибках. Эти сведения могут пригодиться при отладке сайта.

Для остановки сервера вещания следует нажать комбинацию клавиш <Ctrl>+<C> или <Ctrl>+<Break>.

31.2. Бэкенд: вещаемые события и оповещения

31.2.1. Вещаемые события

Обычно отправка данных клиентам посредством вещания реализуется *вещаемыми событиями*. Вещаемое событие создается так же, как и обычное (см. *главу 22*).

Чтобы превратить обычное событие в вещаемое, достаточно:

1. Реализовать в его классе интерфейс Illuminate\Contracts\Broadcasting\ShouldBroadcast.

   В новом классе события, созданном командой make:event утилиты artisan, этот интерфейс уже импортирован, и его остается лишь дописать к объявлению класса.

2. В методе broadcastOn() — вернуть объект канала, через который будет пересылаться текущее вещаемое событие.

Этот метод также присутствует во вновь созданном классе и изначально возвращает объект закрытого канала с именем `channel-name`.

В листинге 31.1 показан код класса вещаемого события `App\Events\BbAdded`, которое сообщает о добавлении нового объявления, содержит само добавленное объявление в свойстве `bb` и пересылается по общедоступному каналу `bbs`.

Листинг 31.1. Код класса вещаемого события `App\Events\BbAdded`

```
namespace App\Events;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;
class BbAdded implements ShouldBroadcast {
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $bb;

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

    public function broadcastOn() {
        return new Channel('bbs');
    }
}
```

Выполнить отправку такого события можно методом `dispatch()` или функцией `event()`, используемыми для генерирования обычных событий (подробности — в *разд. 21.2.4.3*);

```
use App\Events\BbAdded;
. . .
BbAdded::dispatch($bb);
```

Также можно использовать функцию-хелпер `broadcast()`, имеющую тот же формат вызова, что и функция `event()`:

```
broadcast(new BbAdded($bb));
```

Объект предназначенного к отправке события, возвращаемый этой функцией, поддерживает метод `toOthers()`, указывающий отправить событие всем клиентам, кроме текущего:

```
broadcast(new BbAdded($bb))->toOthers();
```

Это может пригодиться в случаях, когда фронтенд отправляет бэкенду какие-либо данные, которые следует сохранить в базе и переслать всем клиентам в составе вещаемого события, и одновременно выводит эти же данные на странице. Если бы фронтенд, от-

правивший данные, получил их вместе с событием, он бы также вывел их на экран, и данные оказались бы выведены дважды.

Функциональность, необходимая для выполнения метода `toOthers()`, реализована в трейте `Illuminate\Broadcasting\InteractsWithSockets`, поэтому предварительно следует включить этот трейт в класс события. Впрочем, любой класс события, сгенерированный командой `make:event` утилиты artisan, уже включает этот трейт.

Чтобы отправить вещаемое событие всем клиентам, кроме текущего, фреймворк должен знать идентификатор сокета текущего клиента. Как передать этот идентификатор серверу, будет рассказано в *разд. 31.4.1*.

При отправке вещаемого события его объект сериализуется и записывается в очередь. Далее обработчик отложенных заданий извлекает объект из очереди и передает для обработки службе вещания.

Вещаемое событие отправляется по каналу, заданному в методе `broadcastOn()`. По умолчанию оно получает имя, совпадающее с полным путем к его классу (например: `App\Events\BbAdded`).

У вещаемого события можно указать дополнительные параметры и изменить его поведение, объявив в его классе следующие общедоступные свойства и методы:

- `afterCommit` — свойство. Если `true`, вещаемое событие будет отправлено только после завершения всех транзакций в базах данных. Если `false`, событие может быть отправлено в произвольный момент времени, в том числе и до завершения всех транзакций. По умолчанию: `false`;
- `broadcastAs()` — метод, должен возвращать строку с именем, под которым событие будет отправлено клиентам (вместо изначального имени, совпадающего с полным путем к его классу):

```
class BbAdded implements ShouldBroadcast {
    . . .
    public function broadcastAs() {
        return 'bb-added';
    }
}
```

  По умолчанию вместе с вещаемым событием отправляется JSON-объект, хранящий значения всех общедоступных свойств объекта этого события;
- `broadcastWith()` — метод, должен возвращать ассоциативный массив со значениями, который будет преобразован в JSON-объект, отправляемый вместе с событием. Применяется в том случае, если свойства события хранят большие объекты, которые необязательно пересылать клиентам целиком, или конфиденциальные данные. Пример:

```
class BbAdded implements ShouldBroadcast {
    . . .
    public function broadcastWith() {
        return ['id' => $this->bb->id, 'title' => $this->bb->title,
                'content' => $this->bb->content,
                'price' => $this->bb->price];
    }
}
```

- `queue` — свойство, задает имя очереди, в которое будет записано событие. Если не объявлено, событие будет помещено в очередь по умолчанию. К сожалению, службу очередей указать невозможно — все вещаемые события помещаются в очереди службы по умолчанию;
- `broadcastQueue()` — метод, должен возвращать имя очереди, в которую будет записано событие. Если не объявлено, имя очереди будет извлечено из свойства `queue`, а если и оно не объявлено, событие будет помещено в очередь по умолчанию;
- `broadcastWhen()` — метод. Если он вернет `true`, событие будет отправлено клиентам посредством вещания, если `false` — не будет. Может пригодиться, если нужно отправлять или не отправлять событие в зависимости от какого-либо условия. Пример:

```
class BbAdded implements ShouldBroadcast {
    . . .
    public function broadcastWhen() {
        return $this->bb->price > 100000;
    }
}
```

Наконец, можно отправить вещаемое событие напрямую, без помещения в очередь (что ускорит его отправку, но может снизить отзывчивость сайта). Для этого достаточно добавить в его класс интерфейс `Illuminate\Contracts\Broadcasting\ShouldBroadcastNow` вместо `ShouldBroadcast`. Пример:

```
. . .
use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;
class BbAdded implements ShouldBroadcastNow {
    . . .
}
```

31.2.1.1. Вещаемые события моделей

Чтобы реализовать вещание событий, возникающих в моделях (см. *разд. 22.3*), на каждое из событий модели, предназначенных к вещанию, придется создавать класс события, делать его вещаемым и связывать с событием модели. Если моделей в проекте много, это отнимет много времени.

Поэтому Laravel предоставляет средства, позволяющие реализовать вещание событий модели минимумом кода. Разработчику понадобится лишь указать каналы, по которым будут отправляться события, а фреймворк сделает все остальное.

Чтобы дать модели возможность отправлять вещаемые события, достаточно выполнить следующие действия:

1. Включить в класс модели трейт `Illuminate\Database\Eloquent\BroadcastsEvents`.
2. Объявить в классе модели общедоступный метод `broadcastOn()`. Он должен принимать в качестве единственного параметра строковое обозначение возникшего в модели события: `'created'` (создана новая запись), `'updated'` (изменена существующая запись), `'deleted'` (запись удалена), `'trashed'` (запись подвергнута «мягкому» удалению) или `'restored'` (восстановлена запись, ранее подвергнутая «мягкому» удалению).

Метод должен возвращать массив каналов, по которым будет отправлено вещаемое событие. В качестве отдельного элемента этого массива можно указать:

- непосредственно объект канала:

```
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Broadcasting\Channel;
class Bb extends Model {
    use BroadcastsEvents;
    . . .
    public function broadcastOn($event) {
        return [new Channel('models')];
    }
}
```

- объект модели — в этом случае отправка события будет выполнена по *закрытому* каналу с именем формата: *<путь к классу модели>.<ключ записи>*. В *пути к классу модели* символы обратного слеша будут заменены точками. Например, при изменении объявления с ключом 26 событие будет отправлено по каналу `App.Models.Bb.26`. Пример:

```
public function broadcastOn($event) {
    return [$this];
}
```

  Можно указать другое имя канала, переопределив метод `broadcastAs()` (будет описан далее).

  Получить имя канала, сформированное на основе объекта модели, можно вызовом метода `broadcastChannel()` модели:

```
public function broadcastOn($event) {
    return [new Channel($this->broadcastChannel())];
}
```

События одного типа можно отправлять по одному каналу, а события другого типа — по другому. Если вернуть «пустой» массив, событие вообще не будет отправлено. Пример отправки события `created` по одноименному каналу, события `updated` — по каналам `models` и `updates` и подавление отправки событий остальных типов:

```
public function broadcastOn($event) {
    if ($event == 'created') {
        return [new Channel('created')];
    } elseif ($event == 'updated') {
        return [new Channel('models'), new Channel('updates')];
    } else {
        return [];
    }
}
```

Вещаемое событие модели представляет собой объект класса `Illuminate\Database\Eloquent\BroadcastableModelEventOccurred`. По умолчанию событие получает имя фор-

мата *<имя класса модели><имя события модели>*, записанное в стиле PascalCase. Так, при создании нового объявления отправляется событие с именем `BbCreated`, а при правке — `BbUpdated`. JSON-объект, пересылаемый с событием, получит свойство `model`, хранящее объект модели.

Изменить это поведение по умолчанию можно, объявив в классе модели такие методы:

- `broadcastAs()` — общедоступный, задает имя канала, используемое при получении из метода `broadcastOn()` объекта модели (вместо объекта канала). Должен принимать единственным параметром строковое имя события и возвращать строку с именем канала. Если метод вернет `null`, будет использовано имя канала, генерируемое по умолчанию. Пример:

```
class Bb extends Model {
    . . .
    public function broadcastAs($event) {
        if ($event == 'created') {
            return 'bb.created';
        } elseif ($event == 'updated') {
            return 'updbb';
        } else {
            return null;
        }
    }
}
```

- `broadcastWith()` — общедоступный, задает структуру JSON-объекта, отправляемого с событием. Должен принимать единственным параметром строковое имя события и возвращать ассоциативный массив, на основе которого и будет создан JSON-объект. Пример:

```
class Bb extends Model {
    . . .
    public function broadcastWith($event) {
        return ['model_id' => $this->id,
                'updated_at' => $this->updated_at];
    }
}
```

- `newBroadcastableEvent()` — защищенный, создает объект вещаемого события. Должен принимать единственным параметром строковое имя события и возвращать объект класса `BroadcastableModelEventOccurred`, представляющий событие. Конструктор этого класса вызывается в формате:

  `BroadcastableModelEventOccurred(`*<объект модели>*`, `*<имя события>*`)`

  Класс `BroadcastableModelEventOccurred` поддерживает метод `dontBroadcastToCurrentUser()`, предписывающий отправить текущее событие всем клиентам, кроме текущего:

```
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred;
class Bb extends Model {
    . . .
```

```
        public function newBroadcastableEvent($event) {
            return (new BroadcastableModelEventOccured($this, $event))
                    ->dontBroadcastToCurrentUser();
        }
    }
```

31.2.2. Вещаемые оповещения

Для отправки данных посредством вещания также можно воспользоваться *вещаемыми оповещениями* (*см. главу 24*).

Чтобы превратить обычное оповещение в вещаемое, достаточно:

1. В массив, возвращаемый методом `via()` класса оповещения, — добавить элемент `'broadcast'`.
2. В классе оповещения — объявить метод `toBroadcast(<`*оповещаемый объект*`>)`. В качестве параметра он должен принимать *объект*, которому отправляется оповещение, — обычно это объект модели `User`, т. е. зарегистрированный пользователь. В качестве результата этот метод должен возвращать объект класса `Illuminate\Notifications\Messages\BroadcastMessage`, конструктор которого имеет следующий формат вызова:

   `BroadcastMessage(<`*ассоциативный массив с отправляемыми данными*`>)`

   Данные, присутствующие в заданном *массиве*, будут отправлены с оповещением в виде JSON-объекта.

   Если метод `toBroadcast()` в классе оповещения отсутствует, для формирования отправляемых данных будет использован метод `toArray()`.

Интерфейс `ShouldQueue` в классе оповещения реализовывать необязательно.

В листинге 31.2 показан код класса вещаемого оповещения `App\Notifications\BbAdded`, которое сообщает о добавлении нового объявления и содержит ключ добавленного объявления в свойстве `bbId`.

Листинг 31.2. Код класса вещаемого оповещения `App\Notifications\BbAdded`

```
namespace App\Notifications;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\BroadcastMessage;
class BbAdded extends Notification {
    use Queueable;

    public $bbId;

    public function __construct($bb) {
        $bbId = $bb->id;
    }
```

```
    public function via($notifiable) {
        return ['broadcast'];
    }

    public function toBroadcast($notifiable) {
        return new BroadcastMessage(['id' => $this->bbId]);
    }
}
```

Отправить такое оповещение текущему пользователю можно обычным способом:

```
use Illuminate\Support\Facades\Auth;
use App\Notifications\BbAdded;
. . .
Auth::user()->notify(new BbAdded($bb));
```

Вещаемые оповещения отправляются по закрытому каналу с именем формата *<путь к классу оповещаемого объекта>.<ключ>*, причем в *пути* вместо обратных слешей используются точки. Например, если вещаемое оповещение отправляется зарегистрированному пользователю — объекту класса `App\Models\User` — с ключом 3, то канал будет иметь имя `App.Models.User.3`.

Можно дать этом каналу другое имя. Достаточно объявить в классе оповещаемого объекта общедоступный метод `receivesBroadcastNotificationsOn()`, возвращающий строковое имя канала. Так, если оповещения отправляются зарегистрированным пользователям, т. е. объектам класса модели `App\Models\User`, нужно объявить метод в этом классе. Пример:

```
class User extends Authenticatable {
    . . .
    public function receiveBroadsactNotificationsOn() {
        return 'users.' . $this->id;
    }
}
```

Каждое вещаемое оповещение имеет тип. По умолчанию он совпадает с полным путем к его классу. Можно указать другой тип, объявив в классе оповещения общедоступный метод `broadcastType()`, возвращающий строку с типом оповещения. Пример:

```
class BbAdded extends Notification {
    . . .
    public function broadcastType() {
        return 'bb-added';
    }
}
```

Класс `BroadcaseMessage` поддерживает методы `onConnection()` и `onQueue()`, задающие соответственно имя службы очередей и имя очереди, куда будет предварительно помещено оповещение, и описанные в *разд. 25.2.2*:

```
public function toBroadcast($notifiable) {
    return (new BroadcastMessage(['id' => $this->bbId]))
              ->onConnection('database')->onQueue('broadcasting');
}
```

31.3. Бэкенд: каналы вещания

Каналов вещания, по которым клиентам будут отправляться вещаемые события и оповещения, может быть создано произвольное количество. Каналы бывают трех типов.

31.3.1. Общедоступные каналы вещания

К *общедоступному каналу вещания* может подключиться любой клиент.

Общедоступный канал представляется объектом класса `Illuminate\Broadcasting\Channel`, чей конструктор вызывается в формате `Channel(<имя канала>)`. *Имя канала* может быть выбрано произвольно.

Если сайт реализует вещание по небольшому количеству каналов, их имена можно сделать состоящими из одного слова (как показано в листинге 31.1). В противном случае удобнее составлять имена каналов из нескольких слов, разделенных точками или дефисами, создавая своего рода иерархию каналов. Пример:

```
public function broadcastOn() {
    return new Channel('Items.Bbs.Added');
}
```

Имя канала может включать какие-либо сведения о текущем пользователе, записи модели, с которой в текущий момент идет работа (например, ключ). Особенно это характерно для закрытых каналов вещания, о которых речь пойдет далее.

31.3.2. Закрытые каналы вещания

Чтобы подключиться к *закрытому каналу вещания*, клиент должен предварительно войти на сайт и успешно пройти авторизацию.

Закрытый канал представляется объектом класса `Illuminate\Broadcasting\PrivateChannel`. Формат вызова его конструктора схож с форматом вызова конструктора класса `Channel` (см. *разд. 31.3.1*). Пример:

```
class UserEvent implements ShouldBroadcast {
    public $user;
    . . .
    public function broadcastOn() {
        return new PrivateChannel('App.Models.User.' . $this->user->id);
    }
}
```

Для прохождения авторизации фронтенд должен выполнить обычный клиентский запрос по особому маршруту с путем **broadcasting/auth**, передав имя закрытого канала в составе запроса. Получив это имя, фреймворк на основании заданной логики авторизации проверит, имеет ли фронтенд право подключаться к этому каналу.

Упомянутый ранее маршрут создается вызовом у фасада `Illuminate\Support\Facades\Broadcast` метода `routes()`. Вызов этого метода присутствует в провайдере `BroadcastServiceProvider`, который перед реализацией вещания следует раскомментировать (см. *разд. 31.1.2*). Маршрут добавляется в список веб-маршрутов.

Логика авторизации закрытых каналов записывается в модуле routes\channels.php в виде набора *маршрутов каналов*. Каждый маршрут формируется вызовом метода `channel()` фасада `Illuminate\Support\Facades\Broadcast` в формате:

```
channel(<шаблонное имя>, <анонимная функция>|<путь к классу канала>[,
                         <настройки>=[]])
```

Шаблонное имя канала записывается аналогично шаблонному пути (см. *главу 8*). В нем можно указать URL-параметры для извлечения каких-либо значений, переданных фронтендом (например, ключа записи модели, с которой ведется работа, или ключа текущего пользователя).

Вторым параметром методу `channel()` можно передать:

- *анонимную функцию*, выполняющую авторизацию. В качестве первого параметра она должна принимать объект текущего пользователя, а остальными параметрами — значения URL-параметров, созданных в *шаблонном имени*. Возвращать она должна `true`, чтобы разрешить фронтенду подключаться к каналу, и `false` — чтобы запретить подключение.

  Пример маршрута, разрешающего подключение к каналу `App.Models.User.<ключ пользователя>` только в том случае, если переданный *ключ пользователя* совпадает с ключом текущего пользователя (т. е. если к каналу подключается текущий пользователь):

  ```
  use Illuminate\Support\Facades\Broadcast;
  Broadcast::channel('App.Models.User.{id}', function ($user, $id) {
      return $user->id == $id;
  });
  ```

  Пример маршрута, разрешающего подключаться к каналу `bb.<ключ объявления>`, только если объявление с заданным *ключом* принадлежит текущему пользователю:

  ```
  Broadcast::channel('bb.{id}', function ($user, $bbId) {
      return $user->id == Bb::findOrNew($bbId)->user->id;
  });
  ```

  В маршрутах канала поддерживается внедрение моделей:

  ```
  use App\Models\Bb;
  Broadcast::channel('bb.{bb}', function ($user, Bb $bb) {
      return $user->id == $bb->user->id;
  });
  ```

- полный *путь к классу канала* в виде строки. Этот класс канала и реализует логику авторизации.

  Новый класс канала создается подачей команды:

  ```
  php artisan make:channel <имя класса канала>
  ```

  Класс канала объявляется в пространстве имен `App\Broadcasting` (соответствующая папка создается автоматически) и не является ничьим подклассом. Он содержит два метода:

  - конструктор — применяется для получения нужных для работы объектов посредством внедрения зависимостей. Изначально «пуст»;

- `join()` — реализует логику авторизации по тем же принципам, что и анонимная функция, указываемая в вызове метода `channel()` фасада `Broadcast`. Изначально «пуст» и принимает единственный параметр — объект текущего пользователя.

В листинге 31.3 показан код класса канала `App\Broadcasting\BbChannel`, разрешающего подключаться к каналу `bb.<ключ объявления>` только пользователю, оставившему объявление с заданным *ключом*.

Листинг 31.3. Код класса канала `App\Broadcasting\BbChannel`

```
namespace App\Broadcasting;
use App\Models\User;
use App\Models\Bb;
class BbChannel {
    public function __construct() { }

    public function join(User $user, Bb $bb) {
        return $user->id == $bb->user->id;
    }
}
```

Вот маршрут, связывающий канал `bb.<ключ объявления>` с только что объявленным классом:

```
use App\Broadcasting\BbChannel;
Broadcast::channel('bb.{bb}', BbChannel::class);
```

Закрытые каналы доступны только пользователям, выполнившим вход. Для выполнения аутентификации пользователей фреймворк использует страж, указанный в настройках как применяемый по умолчанию (см. *разд. 13.1*). Можно указать другой страж, передав методу `channel()` с параметром *настройки* ассоциативный массив, содержащий элемент `guard`, значением которого должен быть массив с нужным стражем. Пример:

```
Broadcast::channel('bb.{bb}', function ($user, Bb $bb) {
    return $user->id == $bb->user->id;
}, ['guard' => ['admin']]);
```

Следует отметить, что окончательное имя закрытого канала, используемое низкоуровневыми инструментами фреймворка, формируется в формате `private-<имя, указанное в маршруте>`. Так, в приведенном ранее примере закрытый канал получит окончательное имя `private-bb.<ключ объявления>`.

31.3.3. Каналы присутствия

Каналы присутствия обычно применяются в чатах и подобного рода веб-службах. По ним рассылаются уведомления о подключении к чату очередного пользователя и о его отключении. Подобно закрытым каналам, перед подключением они требуют от фронтенда пройти авторизацию.

Как только пользователь успешно проходит авторизацию перед подключением к каналу присутствия, он считается присоединившимся к каналу, по которому тотчас рассы-

лается соответствующее уведомление. Аналогичное уведомление рассылается по каналу присутствия, когда пользователь отключается от канала (при переходе на другую страницу или программно, средствами фронтенда). Вместе с этими уведомлениями пересылается JSON-объект со сведениями о пользователе, предоставленными бэкендом.

Список пользователей, подключившихся к каналу присутствия, хранится службой вещания. Если в качестве таковой используется Laravel Websockets, в его настройках, в параметрах соответствующего приложения, нужно включить поддержку каналов присутствия, дав настройке `enable_client_messages` значение `true` (подробности — в *разд. 31.1.3*).

Авторизация фронтенда перед подключением к каналу присутствия выполняется почти так же, как и в случае закрытого канала, — написанием соответствующего маршрута канала с анонимной функцией, реализующей авторизацию, или классом канала. Единственное исключение: чтобы разрешить пользователю подключиться к каналу, нужно вернуть в качестве результата ассоциативный массив со сведениями о пользователе. Пример маршрута, проверяющего, может ли текущий пользователь подключаться к комнате чата с заданным ключом, вызывая заранее объявленный в классе модели метод `canJoinRoom()`:

```
Broadcast::channel('Chat.Room.{room_id}', function ($user, $room_id) {
    if ($user->canJoinRoom($room_id)) {
        return ['userId' => $user->id, 'userName' => $user->name];
    } else {
        return false;
    }
});
```

Возвращенные сведения о пользователе будут сохранены и в дальнейшем отправлены клиентам в составе уведомлений в виде JSON-объекта.

Канал присутствия представляется объектом класса `Illuminate\Broadcasting\PresenceChannel`. Формат вызова его конструктора схож с форматом вызова конструктора класса `Channel` (см. *разд. 31.3.1*). Пример вещаемого события, сообщающего фронтенду о подключении нового пользователя к комнате с указанным ключом:

```
class UserJoined implements ShouldBroadcast {
    public $roomId;
    . . .
    public function broadcastOn() {
        return new PresenceChannel('Chat.Room.' . $this->roomId);
    }
}
. . .
broadcast(new App\Events\UserJoined(1))->toOthers();
```

Окончательное имя канала присутствия формируется в формате `presence-<`*имя, указанное в маршруте*`>`. Так, в приведенном ранее примере канал присутствия получит окончательное имя `presence-Chat.Room.<`*ключ комнаты*`>`.

31.4. Фронтенд: прослушивание каналов вещания

Для прослушивания каналов вещания и получения поступающих по ним вещаемых событий и оповещений удобно применять клиентскую JavaScript-библиотеку *Laravel Echo*, написанную командой разработчиков Laravel.

31.4.1. Использование Laravel Echo

Сначала необходимо установить саму библиотеку Laravel Echo, набрав команду:

```
npm install laravel-echo --save-dev
```

Командный ключ `--save-dev` указывает занести устанавливаемую библиотеку в файл package.json — в список библиотек, используемых лишь при разработке. При обработке файлов веб-сценариев пакет Laravel Mix включит код этой библиотеки в файл vendor.js (подробности о Laravel Mix — в *разд. 17.5*).

Далее следует установить дополнительную библиотеку pusher.js, которая необходима для работы со службами Pusher Channels и Ably и также нужна лишь при разработке:

```
npm install pusher-js --save-dev
```

Код подключения к службам вещания записывается в модуле resources\js\bootstrap.js. Изначально там присутствует код, выполняющий подключение к службе Pusher Channels и закомментированный. Его можно использовать как основу для написания своего кода.

Сначала следует импортировать из библиотеки Laravel Echo класс `Echo`, реализующий работу с каналами вещания:

```
import Echo from 'laravel-echo';
```

а из библиотеки pusher.js — класс `Pusher`, выступающий посредником между классом `Echo` и службами вещания Pusher Channels и Ably:

```
import Pusher from 'pusher-js';
window.Pusher = Pusher;
```

Далее следует создать объект класса `Echo`, передав ему служебный объект с параметрами подключения к службе вещания. Поддерживаются параметры:

- `broadcaster` — строковое обозначение библиотеки, используемой для подключения. Всегда следует указывать обозначение `'pusher'`;
- `key` — ключ пользователя службы Pusher Channels;
- `forceTLS` — если `true`, подключение к службе всегда будет выполняться по защищенному протоколу, если `false` — только если страница была загружена по защищенному протоколу.

Следующий параметр задается только при использовании службы Pusher Channels:

- `cluster` — обозначение кластера службы;

Пример:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: '*** Мой ключ пользователя Pusher Channel ***',
    cluster: 'mt1',
    forceTLS: true
});
```

Следующие параметры используются при работе с сервером вещания Laravel Websockets:

- `key` — ключ пользователя. Необходимо указать ключ из настроек приложения, используемого в качестве точки входа используемой службы вещания (см. *разд. 31.1.3*);
- `wsHost` — интернет-адрес сервера вещания;
- `wsPort` — номер TCP-порта, через который происходит обмен данными с сервером вещания;
- `disableStats` — если `false`, сервер вещания будет вести статистику работы, если `true` — не будет (по умолчанию: `false`).

Пример:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: '12345',
    wsHost: window.location.hostname,
    wsPort: 6001,
    forceTLS: false,
    disableStats: true,
});
```

Следующий параметр используется при работе со службой Ably:

- `encrypted` — если `false`, обмен данными с сервером вещания службы будет выполняться по незащищенному протоколу, если `true` — по защищенному (по умолчанию: `false`).

Также поддерживаются параметры `wsHost`, `wsPort` и `disableStats`.

Пример:

```
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,
});
```

Остальные параметры используются в специфических случаях:

- `client` — готовый объект библиотеки pusher.js, используемый для связи со службой вещания, если таковой был создан ранее. Если параметр не указан, Laravel Echo создаст для своих нужд новый объект соответствующей библиотеки. Пример:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
    broadcaster: 'pusher',
    key: '*** Мой ключ пользователя Pusher Channel ***'
    cluster: 'mt1',
    forceTLS: true
}

window.Echo = new Echo({
    ...options,
    client: new Pusher(options.key, options)
});
```

- `namespace` — имя пространства имен, в котором объявлены классы вещаемых событий, в виде строки (по умолчанию: `App\Events`);
- `authEndpoint` — путь к действию контроллера бэкенда, выполняющему авторизацию для доступа к закрытым каналам и каналам присутствия. По умолчанию: **/broadcasting/auth**.

Если выполняется рассылка вещаемых событий и оповещений всем клиентам, кроме текущего, сайту следует передать идентификатор сокета текущего клиента. Отправить его можно в клиентском запросе, запускающем действие, которое отправит вещаемое событие, в заголовке `X-Socket-ID`. Сам идентификатор можно получить, вызвав у объекта класса `Echo` метод `socketId()`. Пример:

```
async function () {
    fetch( . . . , {
        . . .
        headers: {
            'X-Socket-ID': window.Echo.socketId()
        }
    });
}();
```

31.4.2. Прослушивание общедоступных каналов

Для подключения к общедоступному каналу применяется метод `channel(<`*имя канала*`>)` класса `Echo`. В качестве результата он возвращает объект общедоступного канала, к которому было выполнено подключение.

Чтобы запустить прослушивание канала на предмет передачи по нему вещаемых событий, представленных классом с заданным *именем*, следует вызвать метод `listen()`, поддерживаемый объектом канала:

```
listen(<имя класса вещаемого события>, <анонимная функция>)
```

Имя класса вещаемого события указывается без пространства имен. Laravel Echo сама добавит к нему пространство имен, заданное в параметре подключения `namespace` (см. *разд. 31.4.1*).

Как только по прослушиваемому каналу поступит событие с заданным именем, будет вызвана указанная *анонимная функция*. В качестве единственного параметра она получит объект события, созданный на стороне бэкенда в методе `broadcastOn()` класса события.

Пример подключения к каналу `bbs` с целью получения вещаемого события `BbAdded` (см. листинг 31.1) и извлечения из объекта этого события названия товара и его цены:

```
const chlBbs = window.Echo.channel('bbs');
chlBbs.listen('BbAdded', (evt) => {
    const title = evt.bb.title;
    const price = evt.bb.price;
    . . .
});
```

Можно прослушивать канал с целью получения произвольного количества событий:

```
chlBbs.listen('BbAdded', ... );
chlBbs.listen('BbUpdated', ... );
chlBbs.listen('BbDeleted', ... );
```

Поскольку метод `listen()` в качестве результата возвращает текущий объект канала, его вызовы можно писать цепочкой:

```
chlBbs.listen('BbAdded', ... )
      .listen('BbUpdated', ... )
      .listen('BbDeleted', ... );
```

Если событию было дано другое имя (в методе `broadcastAs()` класса события, подробности — в *разд. 31.2.1*), это имя следует указать в вызове метода `listen()`, *обязательно предварив точкой*, — чтобы Laravel Echo не добавила к нему пространство имен:

```
chlBbs.listen('.bb-added', ... );
```

Аналогично с начальной точкой указываются полные пути к классам событий, объявленных в пространствах имен, отличных от заданного по умолчанию:

```
chlBbs.listen('.App\\BroadcastEvents\\BbAdded', ... );
```

Для прослушивания канала на предмет передачи по нему вещаемого оповещения следует использовать метод `notification(<`*анонимная функция*`>)`. Заданная *анонимная функция* будет выполнена в случае получения по каналу любого оповещения. В качестве параметра она получит объект оповещения, который вдобавок к свойствам, объявленным в его классе, будет иметь свойство `type`, хранящее строку с типом оповещения. Пример:

```
chlBbs.notification('BbAdded', (evt) => {
    if (evt.type == 'App\\Notifications\\BbAdded') {
        const bbId = evt.id;
        . . .
    }
});
```

Чтобы перестать прослушивать канал на предмет получения события с указанным *именем*, следует вызвать метод `stopListening(<`*имя события*`>)` объекта канала:

```
chlBbs.stopListening('BbDeleted');
```

Чтобы отключиться от канала, следует вызвать у объекта класса `Echo` один из двух следующих методов:

- `leaveChannel(<имя канала>)` — выполняет отключение от общедоступного канала с заданным *именем*:

```
window.Echo.leaveChannel('bbs');
```

- `leave(<имя канала>)` — выполняет отключение от общедоступного канала с заданным *именем*, а также от одноименных закрытого канала и канала присутствия.

31.4.3. Прослушивание закрытых каналов

Подключение к закрытому каналу выполняется вызовом у объекта класса `Echo` метода `private()`, аналогичного методу `channel()`:

```
const userId = 123;
const chlUser = window.Echo.private('App.Models.User.' + userId);
chlUser.listen('BbAddedByThisUser', (evt) => {
    const title = evt.bb.title;
    const price = evt.bb.price;
    . . .
});
```

При прослушивании вещаемых событий моделей (которые по умолчанию вещаются по закрытому каналу, подробности — в *разд. 31.2.1.1*) не забываем предварить их имена точками, чтобы Laravel Echo не добавила к ним пространство имен:

```
const bbId = 321;
window.Echo.private('App.Models.Bb.' + bbId)
           .listen('.BbUpdated', (evt) => {
                const id = evt.model.id;
                . . .
            }
);
```

31.4.4. Прослушивание каналов присутствия

Каналы присутствия прослушиваются аналогично закрытым каналам, за тем исключением, что для подключения используется метод `join()` объекта класса `Echo`, аналогичный методу `channel()`:

```
const roomId = 1;
const chlChat = window.Echo.join('Chat.Room.' + roomId);
chlChat.listen('UserJoined', (evt) => {
    . . .
});
```

Для получения уведомлений о подключении и отключении пользователей следует использовать методы, приведенные далее. Все они поддерживаются объектом канала присутствия, возвращаемого методом `join()`:

- `joining(<анонимная функция>)` — запускает прослушивание текущего канала в ожидании уведомления о подключении нового пользователя. При получении такого

уведомления выполняется заданная *анонимная функция*, которая получит в качестве параметра сведения о пользователе, отправленные бэкендом. Пример:

```
chlChat.joining((user) => {
    const id = user.userId;
    const name = user.userName;
    // Выводим сообщение о подключении нового пользователя
});
```

- `leaving(<анонимная функция>)` — запускает прослушивание текущего канала в ожидании уведомления об отключении пользователя (в результате либо вызова метода `leave()`, либо перехода на другую страницу). При получении такого уведомления выполняется заданная *анонимная функция*, которая получит в качестве параметра сведения о пользователе, ранее сохраненные службой вещания. Пример:

```
chlChat.leaving((user) => {
    const id = user.userId;
    const name = user.userName;
    // Выводим сообщение об отключении пользователя
});
```

- `here(<анонимная функция>)` — запускает прослушивание текущего канала в ожидании уведомлений о подключении и отключении пользователей. Вызываемая *анонимная функция* в качестве параметра получает массив из всех подключившихся пользователей, за исключением текущего. Пример:

```
chlChat.here((users) => {
    const fullUserCount = users.length + 1;
    // Выводим полное количество подключившихся пользователей
});
```

31.4.5. Отправка произвольных уведомлений

Также имеется возможность со стороны клиента рассылать другим клиентам по любому закрытому каналу или каналу присутствия произвольные уведомления. Для этого используется метод `whisper()` объекта канала:

```
whisper(<сигнатура уведомления>, <служебный объект с данными>)
```

Сигнатура уведомления задается в виде строки, должна быть уникальной и максимально точно описывать характер отсылаемого уведомления. *Служебный объект* будет преобразован в формат JSON и отправлен вместе с уведомлением. Пример:

```
txtChatMessage.addEventListener('change', function (evt) {
    chlChat.whisper('typing', {userId: userId});
});
```

Для получения таких уведомлений, относящихся к заданному *типу*, используется метод `listenForWhisper()` объекта канала:

```
listenForWhisper(<сигнатура уведомления>, <анонимная функция>)
```

Выполняющаяся при приеме уведомления *анонимная функция* в качестве параметра получает объект с данными, отправленными вместе с уведомлением. Пример:

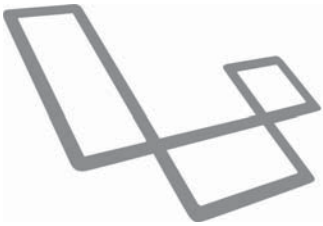
```
chlChat.listenForWhisper('typing', (evt) => {
    divWhisper.textContent = 'Пользователь №' + evt.userId +
                             ' набирает сообщение...';
});
```

31.5. Запуск вещания

Чтобы запустить сайт, реализующий вещание, следует выполнить следующие действия:

1. Запустить обработчик отложенных заданий (см. *разд. 25.6.1*).
2. Если в качестве службы вещания используется сервер Laravel Websockets, запустить этот сервер (см. *разд. 31.1.3*).
3. Запустить сам сайт.

ГЛАВА 32



Команды утилиты artisan

В составе Laravel присутствует утилита командной строки artisan, которая поддерживает большое количество команд, создающих новые программные модули разных типов и выполняющих различные действия над проектом. Команды утилиты artisan запускаются набором командной строки формата:

```
php artisan <командное слово> [<аргументы команды>]
```

Командное слово однозначно указывает, какую команду следует запустить на выполнение (например, командное слово `make:model` запускает команду, создающую новый класс модели). *Аргументы команды* отделяются друг от друга пробелами и делятся на:

- основные — задают важные для работы параметры команды и записываются непосредственно после командного слова (например, у команды создания класса модели основным аргументом является имя создаваемого класса);
- вспомогательные — задают дополнительные параметры, являются необязательными для указания и записываются после основных аргументов. Идентифицируются уникальными именами, обязательно предваряемыми двойными дефисами. Вспомогательный аргумент может принимать значение, которое записывается после его имени через знак равенства. Например, команда выполнения миграций имеет вспомогательные аргументы `--seed` (не принимающий значения) и `--database` (принимающий значение).

Набор поддерживаемых утилитой artisan команд может быть расширен, причем новые команды могут быть реализованы как в виде классов, так и анонимных функций.

32.1. Получение сведений о командах утилиты artisan

Вывести весь список команд, поддерживаемых утилитой, можно набором команды:

```
php artisan list [--raw]
```

По умолчанию выводится список команд, сгруппированных по разделам, например:

```
make
  make:channel          Create a new channel class
  make:command          Create a new Artisan command
```

```
  make:component          Create a new view component class
  make:controller         Create a new controller class
  . . .
migrate
  migrate:fresh           Drop all tables and re-run all migrations
  migrate:refresh         Reset and re-run all migrations
  . . .
model
  model:prune             Prune models that are no longer needed
```

Командный ключ `--raw` указывает вывести простой список команд без группировки:

```
make:channel            Create a new channel class
make:command            Create a new Artisan command
make:component          Create a new view component class
make:controller         Create a new controller class
. . .
migrate:fresh           Drop all tables and re-run all migrations
migrate:refresh         Reset and re-run all migrations
. . .
model:prune             Prune models that are no longer needed
```

Чтобы просмотреть справочную информацию о команде с заданным *именем*, следует набрать одну из следующих команд:

```
php artisan <имя команды> --help
php artisan help <имя команды>
```

32.2. Команды-классы

Команды-классы реализуются в виде классов. Как правило, это достаточно сложные команды, выполняющие множество действий и принимающие, наряду с основными, также и вспомогательные аргументы.

32.2.1. Создание команд-классов

Новый класс команды создается набором следующей команды утилиты artisan:

```
php artisan make:command <имя класса команды> ↳
[--command=<командное слово>]
```

Если указать ключ `--command`, в класс созданной команды будет сразу же записано указанное в этом ключе *командное слово*, запускающее команду. Пример создания класса команды `UserCreate`, создающей нового пользователя, которая будет запускаться командным словом `user:create`:

```
php artisan make:comment UserCreate --command=user:create
```

Созданный класс команды объявляется в пространстве имен `App\Console\Commands` (соответствующая папка создается автоматически) и является производным от класса `Illuminate\Console\Command`.

В классе команды должны присутствовать два следующих защищенных свойства:

- `signature` — строка с описанием формата вызова команды, включающего ее командное слово и перечень поддерживаемых основных и вспомогательных аргументов с их описаниями.

  Формат вызова команды, указываемый в этом свойстве, с одной стороны, сообщает Laravel все необходимые сведения для выполнения этой команды, включая командное слово, а с другой — выводится на экран при вызове справочной информации о команде.

  Изначально свойство хранит строку `'command:name'`. Если при создании класса команды был указан ключ `--command`, свойство будет хранить указанное в этом ключе *командное слово*.

  Более подробно о написании форматов запуска команд будет рассказано далее;

- `description` — краткое описание команды, показываемое при выводе как списка команд утилиты artisan, так и справочной информации о текущей команде. Изначально хранит строку «Command description».

Вся логика команды реализуется в общедоступном методе `handle()`. Он может принимать произвольные параметры, посредством которых будет выполнено получение нужных для работы объектов путем внедрения зависимостей. Также он может возвращать числовой код завершения команды (например, 0, если команда выполнилась успешно). Изначально метод содержит команду возврата кода завершения 0.

В листинге 32.1 показан код команды `App\Console\Commands\UserCreate`, создающей нового зарегистрированного пользователя. Она вызывается в формате:

```
php artisan user:create <адрес электронной почты> [--name=<имя>] ↳
[--password=<пароль>]
```

Адрес электронной почты передается через основной аргумент, а *имя* и *пароль* — через вспомогательные, с именами `--name` и `--password` соответственно. Если *имя* или *пароль* не указаны, команда попросит ввести их вручную. Далее она спросит, создавать ли нового пользователя, и чтобы создать его, следует ввести слово `yes`, а чтобы отказаться от его создания — слово `no` или просто нажать клавишу <Enter>.

Листинг 32.1. Код класса команды `App\Console\Commands\UserCreate`

```
namespace App\Console\Commands;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
class UserCreate extends Command {
    protected $signature = 'user:create ' .
                          '{email : Адрес электронной почты} ' .
                          '{--name= : Имя пользователя} ' .
                          '{--password= : Пароль пользователя}';
```

```
    protected $description = 'Создание нового пользователя';

    public function handle() {
        $email = $this->argument('email');
        $name = $this->option('name');
        if (!$name)
            $name = $this->ask('Введите имя пользователя');
        $password = $this->option('password');
        if (!$password)
            $password =
                $this->secret('Введите пароль пользователя');
        if ($this->confirm('Создать пользователя?')) {
            DB::table('users')
              ->insert(['name' => $name, 'email' => $email,
                        'password' => Hash::make($password)]);
            $this->info('Пользователь ' . $name . ' с адресом ' .
                        $email . ' и паролем ' . $password .
                        ' создан.');
        }
    }
}
```

32.2.2. Описание формата вызова команд

Формат вызова команды, хранящийся в защищенном свойстве `signature` класса этой команды в виде строки, очень важен, поскольку именно из него Laravel узнает, каким командным словом запускается команда и какие аргументы она принимает. Поэтому записывать этот формат следует очень внимательно.

Формат вызова команды записывается в следующем виде:

```
<командное слово> [<основные аргументы через пробел>] ↳
[<вспомогательные аргументы через пробел>]
```

Командное слово записывается как есть.

Все аргументы, как основные, так и вспомогательные, должны иметь уникальные имена, поскольку по ним в коде команды выполняется обращение к этим аргументам с целью получить их значения.

Основные аргументы записываются в следующем формате:

```
{<обозначение аргумента>[ : <описание аргумента>]}
```

Описание аргумента выводится на экран в составе справочной информации о команде и может быть произвольным. *Обозначение аргумента* должно представлять собой:

- имя аргумента — если основной аргумент принимает всего одно значение:

```
user:create {email : Адрес электронной почты}
. . .
php artisan user:create user@site.ru
```

- конструкцию *<имя аргумента>** — если аргумент должен принимать массив значений, при вводе команды набираемых через пробел:

```
user:masscreate {emails* : Адреса электронной почты через пробел}
. . .
php artisan user:masscreate user@site.ru user2@blog.ru user3@forum.ru
```

- конструкцию *<имя аргумента>?* — если основной аргумент не обязателен для указания и *не* имеет значения по умолчанию:

```
users:create {realname? : Настоящее имя (необязательно к указанию)}
```

- конструкцию *<имя аргумента>=<значение по умолчанию>* — если аргумент не обязателен для указания и имеет заданное *значение по умолчанию*, которое он получает, не будучи указанным:

```
user:create {role=author : Привилегии: author (по умолчанию), ↵
editor или admin}
```

Имя каждого вспомогательного аргумента должно предваряться двумя дефисами (например, `--name`). Именно по этим двум дефисам Laravel определяет, что аргумент является вспомогательным.

Вспомогательные аргументы, в частности их *обозначения*, записываются в том же формате, что и основные. *Обозначение аргумента* должно представлять собой:

- имя аргумента — если вспомогательный аргумент не принимает значения и активизирует какую-либо опцию одним своим присутствием:

```
user:create ... {--force : Создать пользователя без запроса}
. . .
php artisan user:create user@site.ru ... --force
```

- конструкцию *<имя аргумента>=* — если вспомогательный аргумент, будучи указанным, должен принимать значение:

```
user:create ... {--name= : Имя пользователя}
. . .
php artisan user:create ... --name=IvanIvanov
```

- конструкцию *<имя аргумента>=<значение по умолчанию>* — если аргумент имеет *значение по умолчанию*, которое он получает, не будучи указанным:

```
user:create {--role=author : Привилегии: author (по умолчанию), ↵
editor или admin}
```

- конструкцию *<имя аргумента>** — если аргумент должен принимать массив значений:

```
user:masscreate ... {--names* : Имена пользователей}
```

  Вспомогательный аргумент, принимающий массив значений, при вводе команды следует набрать столько раз, сколько значений присутствует в передаваемом ему массиве, каждый раз передавая ему очередное значение:

```
php artisan user:masscreate ... --name=IvanIvanov --name=PiotrPetrov
```

У вспомогательного аргумента можно указать сокращенное имя, записав конструкцию формата `--[<сокращенное имя>]|[<имя без двух дефисов>]`. *Сокращенное имя* должно представлять собой одну прописную букву. Пример:

```
user:create ... {--F|force : Создать пользователя без запроса}
```

После чего в вызове команды можно набрать как полное имя вспомогательного аргумента:

```
php artisan user:create user@site.ru --force
```

так и сокращенное, предварив его *одним* дефисом:

```
php artisan user:create user@site.ru -F
```

32.2.3. Получение значений аргументов

Для получения значений аргументов, как основных, так и вспомогательных, необходимо использовать следующие методы, поддерживаемые классом команды:

- `argument(<имя основного аргумента>)` — возвращает значение основного аргумента с заданным *именем*. Если такой аргумент не был указан при вызове команды, возвращается `null`. Пример:

```
public function handle() {
    $email = $this->argument('email');
    . . .
}
```

- `argument()` — возвращает ассоциативный массив со значениями всех основных аргументов, заданных при вызове команды. Ключи элементов этого массива соответствуют именам аргументов, а значения элементов хранят значения аргументов. Пример:

```
$args = $this->argument();
$email = $args['email'];
```

- `arguments()` — аналогичен вызову метода `argument()` без указания параметра;
- `option(<имя вспомогательного аргумента>)` — возвращает значение вспомогательного аргумента с заданным *именем*, которое указывается без двойного дефиса в начале. Если такой аргумент не был указан при вызове команды, возвращается `null`. Пример:

```
public function handle() {
    . . .
    $name = $this->option('name');
    . . .
}
```

- `option()` — возвращает ассоциативный массив со значениями всех вспомогательных аргументов, заданных при вызове команды. Ключи элементов этого массива соответствуют именам аргументов, а значения элементов хранят значения аргументов.

  Для вспомогательного аргумента, не принимающего значения, метод `option()` возвращает `true`, если аргумент был указан, и `false` — в противном случае:

```
user:create ... {--force : Создать пользователя без запроса}
. . .
if ($this->option('force'))
    // Создаем пользователя без вывода запроса
```

- `options()` — аналогичен вызову метода `option()` без указания параметра.

Также могут пригодиться следующие два метода:

- `hasArgument(<имя основного аргумента>)` — возвращает `true`, если основной аргумент с заданным *именем* был указан при запуске команды, и `false` — в противном случае:

```
if ($this->hasArgument('email'))
    // Адрес электронной почты был указан
```

- `hasOption(<имя вспомогательного аргумента>)` — возвращает `true`, если вспомогательный аргумент с заданным *именем* был указан при запуске команды, и `false` — в противном случае.

32.2.4. Получение данных от пользователя

Для получения данных от пользователя необходимо использовать следующие методы, поддерживаемые объектом команды:

- `ask(<приглашение>[, <значение по умолчанию>=null])` — выводит на экран заданное *приглашение* и ожидает от пользователя ввода значения. В качестве результата возвращает введенное значение. Если пользователь в ответ просто нажал клавишу <Enter>, возвращает указанное *значение по умолчанию*. Пример:

```
$name = $this->ask('Введите имя пользователя', 'Vasya Pupkin');
```

- `secret()` — то же самое, что и `ask()`, только не выводит на экран набираемые пользователем символы. Обычно применяется при вводе паролей;

- `confirm(<сообщение>[, <значение по умолчанию>=false])` — выводит на экран заданное *сообщение* и предлагает ответить на него положительно, введя слово `yes`, или отрицательно, введя слово `no`. Возвращает `true`, если пользователь ответил утвердительно, и `false` — в противном случае. Пример:

```
if ($this->confirm('Создать пользователя?'))
    // Пользователь ответил утвердительно. Создаем пользователя.
```

  Можно указать *значение по умолчанию*, которое будет возвращено, если пользователь вместо ответа просто нажал клавишу <Enter>:

```
if ($this->confirm('Создать пользователя?', true))
    // Пользователь ответил утвердительно. Создаем пользователя.
```

- `choice()` — выводит заданные *приглашение* и перечень доступных для выбора пунктов и предлагает пользователю выбрать нужный пункт или пункты (если доступен множественный выбор).

  Перечень пунктов выводится в виде таблицы из двух столбцов: индекса пункта и его содержания. Чтобы выбрать нужный пункт, пользователю достаточно ввести его индекс и нажать клавишу <Enter>. Если активен множественный выбор, для выбора нескольких пунктов нужно указать их индексы через запятую.

  Если можно выбрать лишь один пункт, метод возвращает выбранный пункт в виде строки, если доступен множественный выбор — массив с выбранными пунктами.

  Формат вызова метода:

```
choice(<приглашение>, <массив пунктов>[,
       <пункты по умолчанию>=null[,
```

```
<допустимое количество попыток выбора>=null[,
<множественный выбор?>=false]]])
```

Пункты, присутствующие в заданном *массиве*, должны представлять собой строки.

Пункты по умолчанию будут возвращены методом, если пользователь вместо выбора пунктов просто нажмет клавишу <Enter>. Здесь можно указать:

- индекс пункта в виде целого числа или строки, содержащей целое число, — если можно выбрать лишь один пункт;
- строки из индексов пунктов, разделенных запятыми, — если активен множественный выбор;
- `null` — в этом случае метод вернет значение `null`.

Если пользователь допустит ошибку при выборе пунктов (например, вместо целочисленного индекса введет букву или укажет несуществующий индекс), Laravel попросит его повторить выбор. Можно указать целочисленное *допустимое количество попыток выбора*, по истечении которого фреймворк выведет сообщение об ошибке и прервет исполнение команды (значение `null` задает бесконечное количество попыток).

Если дать параметру *множественный выбор* значение `true`, будет активизирован множественный выбор.

Пример вывода списка названий программных платформ, из которых нужно выбрать лишь одну (по умолчанию будет выбрана PHP):

```
$platform = $this->choice('Выберите программную платформу',
                          ['PHP','Python','Node.js','Ruby'], 0);
```

Пример вывода списка программных платформ с возможностью выбора произвольного их количества, причем после двух безуспешных попыток будет выведено сообщение об ошибке и исполнение команды прервется (по умолчанию в списке выбраны платформы Windows и PHP):

```
$platforms = $this->choice('Выберите программные платформы',
                          ['Windows', 'MySQL', 'PHP', 'Python',
                           'Laravel', 'Linux', 'Redis'],
                          '0,2', 2, true);
```

32.2.5. Вывод данных

Для вывода текстовых сообщений пользователю применяются следующие методы, поддерживаемые объектом команды:

- `line(<текст>)` — выводит *текст* сообщения нейтрального плана цветами по умолчанию (белым на черном фоне):

```
$this->line('Новый пользователь записан в таблицу users.');
```

- `info(<текст>)` — выводит *текст* сообщения об успешном выполнении операции зеленым шрифтом:

```
$this->info('Пользователь успешно создан.');
```

- `error(<текст>)` — выводит *текст* сообщения об ошибке красным шрифтом;
- `alert(<текст>)` — выводит *текст* сообщения о критической ошибке желтым шрифтом в рамке, состоящей из желтых символов звездочки;
- `warn(<текст>)` — выводит *текст* предупреждения желтым шрифтом;
- `comment(<текст>)` — выводит *текст* комментария желтым шрифтом;
- `question(<текст>)` — выводит *текст* вопроса голубым шрифтом;
- `newline([<количество>=1])` — выводит заданное *количество* пустых строк;
- `table(<массив шапки>, <массив строк>)` — выводит таблицу на основе заданных *массивов шапки* и *строк*.

  Массив шапки должен содержать строки, которые будут выводиться в качестве заголовков столбцов таблицы. *Массив строк* должен содержать вложенные массивы, представляющие отдельные строки. Каждый из вложенных массивов должен содержать значения, которые будут выводиться в ячейках соответствующей строки таблицы.

  Выводимая таблица будет иметь рамку, формируемую символами плюса, дефиса и вертикальной черты.

  Пример вывода таблицы программных платформ из двух столбцов, содержащих названия платформ и их типы (клиентская или серверная):

```
$headers = ['Название', 'Тип'];
$platforms = [['PHP', 'Серверная'], ['Python', 'Серверная'],
              ['JavaScript', 'Клиентская']];
$this->table($headers, $platforms);
```

32.2.5.1. Вывод индикатора процесса

При выполнении какого-либо длительного действия можно вывести индикатор, показывающий ход процесса выполнения действия. Сделать это можно двумя способами: простым и сложным.

Простой способ пригодится, если нужно просто перебрать и обработать указанный *массив* или *коллекцию* каких-либо значений. Он реализуется вызовом у объекта команды метода `withProgressBar()`:

```
withProgressBar(<массив или коллекция>, <анонимная функция>)
```

Задаваемая *анонимная функция* должна принимать с единственным параметром очередной элемент указанного *массива* или *коллекции*. Пример:

```
$this->withProgressBar(Bb::all(), function ($el) {
    // Выполняем нужные операции с объявлениями
});
```

Второй способ может оказаться полезным, если требуется более точно контролировать обработку элементов массива или коллекции и вывод индикатора процесса. Для его реализации необходимо:

1. Обратиться к свойству `output` объекта команды, в котором хранится низкоуровневый объект, реализующий вывод на экран.

2. Вызвать у полученного низкоуровневого объекта метод `createProgressBar(<количество делений>)`. В качестве параметра нужно указать *количество делений* шкалы создаваемого индикатора, которое должно быть равным количеству элементарных операций, из которых состоит выполняемое действие (например, если выполняется обработка записей в таблице, в качестве количества делений надо указать количество обрабатываемых записей). В результате метод вернет созданный объект индикатора процесса. Пример:

```
$bar = $this->output->createProgressBar(Bb::count());
```

3. Перед началом выполнения действия — вызвать метод `start()` объекта индикатора процесса, чтобы вывести индикатор на экран и установить его в начальное деление шкалы:

```
$bar->start();
```

   Объект из свойства `output` команды также поддерживает метод `progressStart()`, который имеет тот же формат вызова, что и метод `createProgressBar()`, и заменяет методы `createProgressBar()` и `start()`:

```
$bar = $this->output->progressStart(Bb::count());
```

4. После выполнения очередной операции — вызвать метод `advance([<смещение>=1])` объекта индикатора процесса, который сдвинет шкалу индикатора вперед на заданное *смещение*, выражаемое в количестве делений шкалы:

```
foreach (Bb::all() as $bb) {
    // Выполняем нужные операции с объявлениями
    $bar->advance();
}
```

5. По завершении обработки всех элементов массива или коллекции — вызвать метод `finish()` объекта индикатора процесса, чтобы довести его шкалу до конечного деления:

```
$bar->finish();
```

32.2.6. Вызов из команды других команд

Если из создаваемой команды утилиты artisan требуется вызвать на выполнение другую команду, пригодятся два следующих метода объекта команды:

- `call()` — выполняет команду с заданным *командным словом*, передавая ей основные и вспомогательные аргументы, приведенные в указанном ассоциативном *массиве*, и выводит на экран весь вывод этой команды:

  `call(<командное слово>[, <массив аргументов>=[]])`

  В ассоциативном *массиве аргументов* ключи элементов должны соответствовать именам аргументов (имена вспомогательных аргументов нужно указывать с префиксом в виде двух дефисов), а значения элементов зададут значения аргументов. Последние могут быть как элементарными значениями, так и массивами — если соответствующие аргументы вызываемой команды принимают массивы значений. Если вызываемой команде нужно передать вспомогательный аргумент, не прини-

мающий значения, следует присвоить соответствующему элементу *массива* значение `true`. Пример вызова команды `make:model`:

```
$this->call('make:model',
            ['name' => 'Offer', '--controller' => true]);
```

- `callSilent()` — то же самое, что и `call()`, только весь вывод вызываемой команды, наоборот, *не* выводится на экран.

32.2.7. Регистрация команд-классов

Все команды-классы, объявленные в пространстве имен `App\Console\Commands`, сразу становятся готовыми к выполнению. Однако команды, объявленные в другом пространстве имен, потребуют явной регистрации в проекте, чтобы утилита artisan смогла их выполнить.

Можно зарегистрировать:

- сразу все команды-классы, которые объявлены в пространстве имен, отличном от `App\Console\Commands`.

  Для этого следует открыть модуль с «корневым» классом утилиты artisan `App\Console\Kernel`, найти объявленный в нем защищенный метод `commands()` и добавить в него вызов метода `load(<путь к папке с командами-классами>)`, поддерживаемого тем же классом. Пример регистрации в проекте всех команд-классов, находящихся в папке app\Console\DBCommands (и соответственно в пространстве имен `App\Console\DBCommands`):

```
class Kernel extends ConsoleKernel {
    . . .
    protected function commands() {
        $this->load(__DIR__.'/Commands');
        $this->load(__DIR__.'/DBCommands');
        . . .
    }
}
```

- отдельную команду-класс — добавив путь к ее классу в массив, присвоенный защищенному свойству `commands` того же класса `Kernel`. Пример регистрации команды-класса `App\Console\VerySpecialCommands\Discount`:

```
class Kernel extends ConsoleKernel {
    protected $commands = [VerySpecialCommands\Discount::class];
    . . .
}
```

32.3. Команды-функции

Команды-функции реализуются в виде анонимных функций и практически всегда являются очень простыми, выполняющими одно действие и принимающими лишь основные аргументы.

Команды-функции записываются в виде так называемых *маршрутов команд* в модуле routes\commands.php. Каждый маршрут создается вызовом у фасада `Illuminate\Support\Facades\Artisan` метода `command()`:

`command(<`*`шаблонное командное слово`*`>, <`*`анонимная функция`*`>)`

Шаблонное командное слово пишется по тем же правилам, что и шаблонный путь (см. *главу 8*). В нем можно указать URL-параметры для извлечения значений основных аргументов.

Указываемая *анонимная функция*, собственно, и реализует создаваемую команду. В качестве параметров она получит значения URL-параметров, созданных в *шаблонном командном слове*.

В *анонимной функции* доступна псевдопеременная `this`, ссылающаяся на текущий объект команды. Таким образом, внутри тела этой функции можно вызывать описанные в *разд. 32.2* методы, запрашивающие данные у пользователя, выводящие ему результаты и вызывающие другие команды.

Метод `command()` возвращает объект, представляющий созданную команду-функцию. Этот объект поддерживает метод `describe(<`*`краткое описание команды`*`>)`, задающий *краткое описание команды*.

Пример команды-функции:

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
. . .
Artisan::command('user:create {email}', function ($email) {
    $name = $this->ask('Введите имя пользователя');
    $password = $this->secret('Введите пароль пользователя');
    DB::table('users')->insert(['name' => $name, 'email' => $email,
                                'password' => Hash::make($password)]);
    $this->info('Пользователь ' . $name . ' с адресом ' . $email .
                ' и паролем ' . $password . ' создан.');
})->describe('Создание нового пользователя');
```

Вместо метода `describe()` можно использовать полностью аналогичный ему метод `purpose()`.

В анонимных функциях, указываемых в вызовах метода `command()`, работает внедрение зависимостей. Параметры, которым будут присваиваться объекты, получаемые в результате внедрения зависимостей, должны располагаться *перед* параметрами, в которые будут заноситься значения URL-параметров. Пример:

```
Artisan::command('email:send {user_id}',
             function (SuperMailer $mailer, $user_id) {
                 $mailer->send(User::find($user_id));
             })
       ->describe('Отправка письма пользователю');
```

32.4. Программный вызов команд

Команды утилиты artisan можно вызывать не только из других команд, но и из любых других мест кода сайта (например, из действий контроллера). Это позволяют сделать два следующих метода фасада `Illuminate\Support\Facades\Artisan`:

- `call()` — выполняет команду с заданным *командным словом*, передавая ей указанный ассоциативный *массив аргументов*:

  ```
  call(<командное слово>|<командная строка с аргументами>[,
       <массив аргументов>=[]])
  ```

  Пример:

  ```
  use Illuminate\Support\Facades\Artisan;
  . . .
  Artisan::call('make:model',
                ['name' => 'Offer', '--controller' => true]);
  ```

  Вместо *командного слова* можно указать полную *командную строку с аргументами*:

  ```
  Artisan::call('make:model Offer --controller');
  ```

- `queue()` — то же самое, что и `call()`, только помещает команду в очередь с тем, чтобы ее выполнил обработчик очередей (*отложенная команда*, об очередях рассказывалось в *главе 25*). Это повысит отзывчивость сайта.

  Объект отложенной команды, возвращаемый методом `queue()`, поддерживает методы: `onQueue()`, `onConnection()` и `delay()`, задающие очередь, службу очередей и задержку соответственно и описанные в *разд. 25.2.2*. Пример их использования:

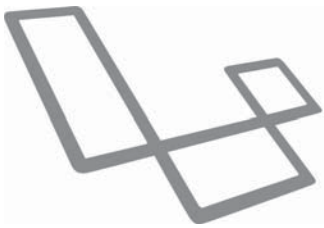
  ```
  Artisan::call('make:model Offer --controller')
         ->onQueue('commands')
         ->onConnection('database');
  ```

32.5. События утилиты artisan

В процессе работы утилиты artisan генерируются следующие события, классы которых объявлены в пространстве имен `Illuminate\Console\Events`:

- `ArtisanStarting` — генерируется при запуске утилиты artisan. Поддерживается свойство `artisan`, хранящее объект класса `Illuminate\Console\Application`, который представляет саму эту утилиту;
- `CommandStarting` — генерируется перед запуском команды утилиты artisan. Поддерживается свойство `command`, которое содержит командное слово, запустившее команду, в виде строки;
- `CommandFinished` — генерируется после выполнения команды. Поддерживаются свойства:
  - `command` — командное слово, запустившее команду, в виде строки;
  - `exitCode` — целочисленный код завершения команды.

ГЛАВА 33



Обработка ошибок

Высокоуровневые инструменты Laravel в случае возникновения нештатной ситуации сами возбуждают исключение, которое отправляет посетителю страницу с соответствующим сообщением об ошибке. Например, если механизм внедрения моделей (см. *разд. 8.5.2*) не найдет в базе данных запись с полученным через URL-параметр ключом, он возбудит исключение `ModelNotFoundException`, отправляющее посетителю сообщение об ошибке с кодом 404.

Также можно отправлять страницы с сообщениями об ошибках самостоятельно, используя функции, которые описывались в *разд. 9.6*, например:

```
abort(404, 'Объявление не найдено');
```

Наконец, фреймворк позволяет как править страницы сообщений об ошибках, так и объявлять свои исключения, которые будут сообщать посетителям об ошибках, специфических для конкретного сайта.

33.1. Настройка веб-страниц с сообщениями об ошибках

Чтобы указать фреймворку использовать другую страницу для уведомления посетителей о возникновении ошибки с определенным кодом статуса, достаточно:

1. Создать в папке resources\views вложенную папку errors (если она не была создана там ранее).
2. Поместить в эту папку файл, хранящий код шаблона страницы с сообщением об ошибке, дав ему имя, совпадающее с кодом статуса этой ошибки (так, шаблон с сообщением об ошибке 404 должен храниться в файле 404.blade.php).

   Впоследствии этот шаблон будет использован для вывода страницы с сообщением об ошибке вместо встроенного в Laravel.

Если необходимо переделать все страницы с сообщениями об ошибках (например, чтобы привести их в соответствие с дизайном сайта), проще всего скопировать их в папку resources\views\errors и соответственно исправить. Скопировать страницы можно подачей команды:

```
php artisan vendor:publish --tag=laravel-errors
```

Будут скопированы следующие файлы:

- 401.blade.php, 403.blade.php, 404.blade.php, 419.blade.php, 429.blade.php, 500.blade.php и 503.blade.php — шаблоны страниц с сообщениями об ошибках соответствующих кодов статуса.

  Все эти шаблоны являются производными от базового шаблона minimal.blade.php;
- minimal.blade.php — базовый шаблон страниц с сообщениями об ошибках. Выводит в центре страницы в строку код ошибки и текстовое сообщение;
- layout.blade.php — альтернативный базовый шаблон страниц с сообщениями об ошибках. Выводит в центре страницы только сообщение об ошибке.

Базовые шаблоны содержат следующие секции:

- `title` — название страницы (выводится в теге `<title>`);
- `code` — числовой код ошибки;
- `message` — текстовое сообщение об ошибке.

Кроме того, в контексте шаблонов страниц с сообщениями об ошибках присутствует переменная `exception`, хранящая объект встроенного в PHP класса `Exception`, который содержит сведения об ошибке.

Разумеется, при необходимости можно переделать эти базовые шаблоны или даже создать новый. Также можно сделать шаблоны страниц с сообщениями об ошибках производными от того же базового шаблона, что и остальные страницы сайта.

Наконец, в папке resources\views\errors можно создать следующие шаблоны:

- 4xx.blade.php — для вывода сообщений об ошибках с кодами статуса 400–499;
- 5xx.blade.php — для вывода сообщений об ошибках с кодами статуса 500–599.

Эти шаблоны будут использованы для вывода сообщений с кодами, для которых отсутствует «собственный» шаблон (например, для вывода сообщения об ошибке 414, если шаблон 414.blade.php отсутствует).

33.2. Создание и использование своих исключений

Если посетителю требуется выводить сообщения о каких-либо специфических для конкретного сайта ошибках (например, о попытке удалить рубрику, в которой есть объявления), имеет смысл объявить свои классы исключений.

33.2.1. Создание своих исключений

Новый класс исключения создается командой:

```
php artisan make:exception <имя класса исключения> [--render] [--report]
```

Командные ключи будут описаны чуть позже, во время рассмотрения методов класса исключения.

Новый класс исключения объявляется в пространстве имен `App\Exceptions` как производный от встроенного в PHP класса `Exception`. Он может содержать два следующих метода:

- `render()` — должен возвращать серверный ответ, содержащий страницу с сообщением об ошибке. В качестве единственного параметра принимает объект текущего запроса.

  Серверный ответ со страницей должен представляться объектом класса `Response`, поскольку только у него можно указать код статуса, соответствующий возникшей ошибке (подробности — в *разд. 9.5.1.2*).

  Если из метода в качестве результата вернуть значение `false`, будет выведена стандартная страница с сообщением об ошибке в коде сайта.

  Если при создании класса указать командный ключ `--render`, будет создан «пустой» метод `render()`, в который останется лишь записать необходимый код.

  Если же ключ `--render` не указывать, будет создан класс вообще без метода `render()`. Тогда при возбуждении исключения этого класса, опять же, будет выводиться стандартная страница с сообщением об ошибке в коде сайта;

- `report()` — должен заносить в журнал запись об ошибке.

  Если из метода в качестве результата вернуть значение `false`, в журнал будет занесена стандартная запись об ошибке в коде сайта.

  Если при создании класса указать командный ключ `--report`, будет создан «пустой» метод `report()`, в который останется лишь вставить необходимый код.

  Если ключ `--report` не указывать, при возбуждении исключения этого класса в журнал будет занесена стандартная запись об ошибке в коде сайта.

  Метод `report()` может принимать произвольные параметры, используемые для получения необходимых для работы объектов путем внедрения зависимостей.

В листинге 33.1 показан код класса исключения `App\Exceptions\RubricNotEmpty`, которое выводит сообщение о том, что рубрика не пуста (содержит объявления). Такое исключение может генерироваться при попытке удаления непустой рубрики.

Листинг 33.1. Код класса исключения `App\Exceptions\RubricNotEmpty`

```
namespace App\Exceptions;
use Exception;
class RubricNotEmpty extends Exception {
    public function render($request) {
        return response()->view('errors.rubricnotempty',
                                ['exception' => $this], 500);
    }
}
```

В листинге 33.2 приведен код шаблона resources\views\errors\rubricnotempty.blade.php для этого исключения.

Листинг 33.2. Код шаблона resources\views\errors\rubricnotempty.blade.php

```
@extends('layouts.app')

@section('title', 'Рубрика не пуста')

@section('content')
    <p>Рубрика &laquo;{{ $exception->getMessage() }}&raquo; содержит
       объявления.</p>
@endsection('content')
```

Можно указать дополнительную информацию, которая будет записана в журнал при возбуждении своего исключения (подробности о занесении записей приведены в *разд. 34.1.2*). Для этого достаточно объявить в классе этого исключения общедоступный метод `context()`. Он не должен принимать параметров и должен возвращать ассоциативный массив, содержащий подлежащие добавлению в журнал данные. Пример указания записать в журнал все данные, полученные от посетителя:

```
class RubricNotEmpty extends Exception {
    . . .
    public function context() {
        return ['userData' => request()->input()];
    }
}
```

33.2.2. Возбуждение своих исключений

Возбудить свое исключение можно посредством PHP-оператора `throw`:

```
use App\Exceptions\RubricNotEmpty;
. . .
public function destroy(Rubric $rubric) {
    if ($rubric->bbs()->exists())
        throw new RubricNotEmpty($rubric->name);
    $rubric->delete();
    . . .
}
```

Также можно использовать следующие функции-хелперы:

- `throw_if()` — возбуждает исключение только в том случае, если заданное *условие* в результате вычисления дает `true`:

  ```
  throw_if(<условие>, <исключение>[, <параметр исключения 1>,
                                    <параметр исключения 2>,
                                    . . .
                                    <параметр исключения n>])
  ```

  Исключение может быть задано:

  - в виде строки с путем к его классу — в таком случае функция `throw_if()` сама создаст объект этого исключения, передав конструктору класса все указанные в ее вызове *параметры*, начиная с третьего:

```
throw_if($rubric->bbs()->exists(), RubricNotEmpty::class,
                                   $rubric->name);
```

- в виде готового объекта исключения:

```
$exception = new RubricNotEmpty($rubric->name);
throw_if($rubric->bbs()->exists(), $exception);
```

- `throw_unless()` — возбуждает исключение только в том случае, если заданное *условие* в результате вычисления дает `false`. Формат вызова такой же, как и у функции `throw_if()`.

33.3. Настройка обработки существующих исключений

Если нужно лишь изменить обработку исключений встроенных в Laravel классов, объявлять свои классы исключений не нужно. Достаточно записать необходимую логику в класс стандартного обработчика исключений `App\Exceptions\Handler`.

По умолчанию в журнал заносятся записи обо всех исключениях, в том числе и созданных самим разработчиком сайта. Если какие-либо исключения регистрировать в журнале не нужно, следует открыть модуль с классом стандартного обработчика исключений `App\Exceptions\Handler` и добавить пути к классам нужных исключений в массив, присвоенный защищенному свойству `dontReport`. Пример:

```
class Handler extends ExceptionHandler {
    protected $dontReport = [
        \Illuminate\Database\Eloquest\ModelNotFoundException::class,
    ];
    . . .
}
```

По умолчанию записи обо всех исключениях заносятся в журнал под уровнем `error` (уровни записей журнала описаны в *разд. 34.1.1*). Однако можно указать свои уровни записей для определенных классов исключений. Для этого следует в классе стандартного обработчика исключений добавить к ассоциативному массиву из защищенного свойства `levels` элементы с ключами, представляющими собой полные пути к классам исключений, и значениями — строковыми обозначениями уровней: `'debug'`, `'info'`, `'notice'`, `'warning'`, `'error'`, `'critical'`, `'alert'` или `'emergency'`. Вместо строковых обозначений можно использовать константы `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `CRITICAL`, `ALERT` и `EMERGENCY` класса `Psr\Log\LogLevel`. Пример:

```
use Psr\Log\LogLevel;
class Handler extends ExceptionHandler {
    protected $levels = [
        \Illuminate\Database\Eloquest\ModelNotFoundException => 'info',
        \Illuminate\Database\Eloquest\RelationNotFoundException =>
                                                    LogLevel::NOTICE,
    ];
    . . .
}
```

По умолчанию в состав дополнительной информации, заносимой в журнал вместе с записью об исключении, входит только ключ текущего пользователя. Однако можно добавить туда произвольные данные. Для этого следует объявить в классе стандартного обработчика исключений защищенный метод `context()`, не принимающий параметров и возвращающий ассоциативный массив со всеми данными, предназначенными для записи в журнал. Пример указания записать в журнал все данные, полученные от посетителя:

```
use Illuminate\Support\Arr;
class Handler extends ExceptionHandler {
    . . .
    protected function context() {
        return Arr::add(parent::context(), 'userdata',
                                           request()->input());
    }
    . . .
}
```

И, наконец, можно полностью переделать обработку существующих исключений, а именно — вывод страниц с сообщениями и (или) занесение в журнал записей о них. Для этого достаточно записать в теле метода `register()` класса стандартного обработчика исключения вызовы следующих методов, поддерживаемых тем же классом:

- `renderable(<`*анонимная функция*`>)` — задает *анонимную функцию*, генерирующую и возвращающую серверный ответ с сообщением о возникновении исключения определенного типа.

  Задаваемая *анонимная функция* должна принимать два параметра: объект возникшего исключения и объект текущего клиентского запроса. У первого параметра в качестве типа следует указать класс исключений, сообщения о которых будут генерироваться этой *функцией*.

  Возвращаемый *функцией* серверный ответ должен представляться объектом класса `Response` (был описан в *разд. 9.5.1.2*).

  Если заданная *анонимная функция* не вернула никакого результата, будет выведена стандартная страница с сообщением об исключении;

- `reportable(<`*анонимная функция*`>)` — задает *анонимную функцию*, заносящую в журнал запись о возникшем исключении определенного типа.

  Задаваемая *анонимная функция* должна принимать с единственным параметром объект возникшего исключения. У параметра в качестве типа следует указать класс исключений, записи о которых будут заноситься этой *функцией* в журнал.

  По умолчанию помимо записи, заносимой указанной *анонимной функцией*, в журнал также будет заноситься стандартная запись об исключении. Отключить занесение стандартной записи можно, либо вернув из *анонимной функции* значение `false`, либо вызвав у результата, возвращенного методом `reportable()`, метод `stop()`.

Пример:

```
use \Illuminate\Database\Eloquest\ModelNotFoundException;
class Handler extends ExceptionHandler {
    . . .
```

```
    public function register() {
        $this->renderable(function (ModelNotFoundException $exc,
                                    $request) {
            return response()->view('errors.modelnotfound.blade.php',
                                    ['exception' => $exc], 500);
        });
        $this->reportable(function (ModelNotFoundException $exc) {
            logger()->warning('Запись не найдена');
        })->stop();
    }
}
```

33.4. Подавление исключений

При возникновении любого исключения, чей класс не указан в массиве из свойства `dontReport` класса стандартного обработчика, в журнал заносится соответствующая запись. Кроме того, любое исключение прерывает обработку текущего клиентского запроса и выводит страницу с сообщением об ошибке.

Однако иногда при возникновении исключения требуется завершить обработку текущего запроса без вывода страницы с сообщением об ошибке (но с занесением записи в журнал). Или, другими словами, произвести *подавление исключений*.

Подавить исключение, представленное заданным *объектом*, проще всего вызовом функции-хелпера `report(<объект исключения>)`. Пример:

```
try {
    // Здесь возможно возникновение исключения SomeException
} catch (SomeException $exc) {
    // Подавляем возникшее исключение.
    // Страница с сообщением об ошибке не появится.
    report($exc);
}
// Продолжаем обработку текущего запроса…
```

Чтобы подавить возникновение исключений в каком-либо фрагменте кода, удобно использовать функцию `rescue()`:

```
rescue(<анонимная функция>[, <значение по умолчанию>=null[,
                             <занести запись в журнал?>=true]])
```

Фрагмент кода, в котором могут возникнуть исключения, оформляется в виде не принимающей параметров *анонимной функции*, которая указывается первым параметром. Эта *функция* может возвращать результат, который, в свою очередь, будет возвращен функцией `rescue()`. Пример:

```
use App\Models\User;
// Пытаемся получить адрес электронной почты заведомо существующего
// пользователя admin
$email = rescue(function () {
    return User::firstWhere('name', 'admin')->email;
});
// Результат: 'admin@bboard.ru'
```

```
// Пытаемся получить адрес электронной почты
// заведомо несуществующего пользователя superadmin
$email = rescue(function () {
    return User::firstWhere('name', 'superadmin')->email;
});
// Результат: null
```

Можно указать *значение по умолчанию*, которое будет возвращено, если при выполнении *анонимной функции* возникнет исключение:

```
$email = rescue(function () {
    return User::firstWhere('name', 'superadmin')->email;
}, 'Таких нет');
// Результат: 'Таких нет'
```

Вместо *значения по умолчанию* можно указать анонимную функцию, которая в качестве параметра примет объект возникшего исключения и вернет результат, который, в свою очередь, будет возвращен функцией `rescue()`:

```
$email = rescue(function () {
    return User::firstWhere('name', 'superadmin')->email;
}, function () {
    return User::firstWhere('name', 'admin')->email;
});
// Результат: 'admin@bboard.ru'
```

Если параметру *занести запись в журнал* дать значение `true`, запись о возникшем исключении будет занесена в журнал, а если дать значение `false` — не будет.

Функция `retry()` делает заданное *количество попыток* выполнить заданную *анонимную функцию* через заданный *интервал времени* (задается в миллисекундах). Если по истечении указанного *количества попыток* заданную *анонимную функцию* не удастся выполнить вследствие возникающего в ней исключения, функция `retry()` завершает работу с возбуждением возникшего исключения. Формат вызова:

```
retry(<количество попыток>, <анонимная функция>[, <интервал времени>=0[,
                            <анонимная функция-прерыватель>=null]])
```

Анонимная функция в качестве параметра должна принимать количество уже сделанных попыток ее выполнить. Пример:

```
$result = retry(3, function ($attempts) {
    . . .
}, 100);
```

Третьим параметром вместо *интервала времени* можно указать анонимную функцию, которая будет принимать в качестве параметров количество уже сделанных попыток выполнить *анонимную функцию* из первого параметра и объект исключения и возвращать значение интервала времени перед следующей попыткой выполнить *анонимную функцию*:

```
$result = retry(3, function ($attempts) {
    . . .
}, function ($attempts, $exc) {
    return $attempts * 100;
});
```

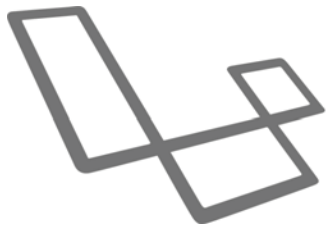
Также первым параметром можно указать массив со значениями интервалов времени между отдельными попытками выполнить *анонимную функцию*:

```
$result = retry([100, 200, 500], function ($attempts) {
    . . .
}, 100);
```

Четвертым параметром можно задать *анонимную функцию-прерыватель*, которая будет выполняться после каждой неудачной попытки вызвать *анонимную функцию* и получить в качестве параметра объект возникшего исключения. Если *прерыватель* вернет значение `false`, функция `retry()` немедленно завершит свою работу. Пример:

```
$result = retry(3, function ($attempts) {
    . . .
}, 100,  function ($exc) {
    return $exc instanceof CriticalException;
});
```

ГЛАВА 34



Журналирование и дополнительные средства отладки

Подсистема *журналирования* ведет подробный журнал работы сайта, который включает в себя записи о возникших ошибках. Этот журнал может храниться как в локальном файле, так и на удаленной веб-службе. Изучая журнал, разработчики смогут быстрее понять, какой фрагмент кода вызывает ошибку, найти и исправить ее.

34.1. Подсистема журналирования

Подсистема журналирования Laravel основана на популярной PHP-библиотеке Monolog (**https://github.com/Seldaek/monolog**).

34.1.1. Настройка подсистемы журналирования

Настройки подсистемы журналирования хранятся в модуле config\logging.php:

- `channels` — хранит список доступных *каналов журналирования*. Список представлен ассоциативным массивом, ключи элементов которого представляют имена каналов, а значениями элементов являются вложенные ассоциативные массивы, хранящие настройки отдельных каналов. Доступны следующие настройки:
  - `driver` — тип службы журналирования, используемой каналом для хранения журнала. Поддерживаются типы:
    - `single` — заносит все записи журнала в один локальный файл;
    - `daily` — заносит записи, созданные в течение одного дня, в отдельный локальный файл. Удаляет все файлы журналов, созданные ранее указанного количества дней;
    - `slack` — использует для ведения журнала веб-службу Slack (**https://slack.com/**);
    - `syslog` — заносит записи в журнал операционной системы;
    - `errorlog` — использует функцию `error_log()`, встроенную в PHP;
    - `monolog` — непосредственно использует библиотеку Monolog;
    - `stack` — служба-агрегатор, объединяющая произвольное количество других служб. Пересылает заносимые в нее записи всем службам, входящим в ее состав.

Конфигурирование библиотеки Monolog и создание своих служб журналирования...

... описывается на странице: **https://laravel.com/docs/9.x/logging**.

- `name` — имя подканала. Если не указано, записи заносятся в подканал с именем, совпадающим с наименованием режима работы сайта: `local` или `production` — берется из настройки `app.env` (подробности — в *разд. 3.4.2.2*);
- `level` — строковое обозначение минимального уровня записей, заносимых в журнал, в виде строки. Записи с более низкими уровнями будут игнорироваться. Поддерживаются уровни:
  - `debug` — уведомления о выполнении каких-либо элементарных действий или вывод результатов каких-либо промежуточных вычислений;
  - `info` — уведомления о выполнении каких-либо действий более высокого уровня (например, о выполнении входа на сайт);
  - `notice` — уведомления более высокой важности (например, о добавлении, правке или удалении записи таблицы);
  - `warning` — сигналы о малозначительных проблемах, не способных нарушить работу сайта (например, об отсутствии в таблице записи с указанным ключом);
  - `error` — сигналы о малозначительных ошибках (например, о сбое в действии контроллера, выводящем второстепенную страницу);
  - `critical` — уведомления о серьезных ошибках, тем не менее не способных нарушить работу всего сайта (неработоспособность подсистемы отправки электронных писем, подсистемы кеширования и т. п.);
  - `alert` — сигналы о критических ошибках (наподобие неработоспособности базы данных);
  - `emergency` — уведомления о полной неработоспособности сайта.

  Вместо строковых обозначений уровней можно использовать константы `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `CRITICAL`, `ALERT` и `EMERGENCY` класса `Psr\Log\LogLevel`.

  Значение настройки берется из локальной настройки `LOG_LEVEL` (изначально: `debug`).

Следующие параметры используются только службами `single` и `daily`:

- `bubble` — если `true`, запись после занесения в журнал будет передана другим каналам для обработки, если `false` — не будет (по умолчанию: `true`);
- `permission` — права на доступ к файлам журнала (по умолчанию: 0644, т. е. владелец имеет право на чтение и запись, остальные — только на чтение);
- `locking` — если `true`, файл журнала перед занесением очередной записи будет блокироваться, чтобы не дать записать что-либо в него другим программам и, таким образом, предотвратить возможное повреждение журнала. Если `false`, журнал блокироваться не будет. По умолчанию: `false`.

Следующий параметр используется только службой `single`:

- `path` — полный путь к файлу журнала (по умолчанию: путь к файлу storage\logs\laravel.log).

Следующие параметры используются только службой `daily`:

- `path` — базовый путь к файлам журнала. Пути к файлам формируются согласно формату *<базовый путь без расширения файла>-<год>-<номер месяца>-<число>.<расширение файла>*. По умолчанию: полный путь к файлу storage\logs\laravel.log;
- `days` — количество дней, в течение которых нужно хранить файлы журналов, в виде целого числа. Файлы, созданные ранее этого количества дней, удаляются. По умолчанию: 14.

Следующие параметры используются только службой `slack`:

- `url` — интернет-адрес для взаимодействия с веб-службой. Значение берется из локальной настройки `LOG_SLACK_WEBHOOK_URL`, изначально отсутствующей в файле .env;
- `username` — строковое имя пользователя веб-службы (по умолчанию: `'Laravel Log'`);
- `emoji` — строковое обозначение эмодзи для прикрепления к заносимым в журнал записям (по умолчанию: `':boom:'`).

Следующие параметры используются только службой `stack`:

- `channels` — массив имен каналов, которые будет объединять текущий канал (по умолчанию: массив с единственным элементом `'single'`);
- `ignore_exceptions` — если `true`, исключения, возникающие в службах, используемых объединяемыми каналами, будут игнорироваться, чтобы остальные каналы смогли занести записи в свои журналы. Если `false`, любое возникшее в службе журналирования исключение прервет работу остальных каналов. По умолчанию: `false`.

Изначально созданы следующие каналы:

- `stack`, `single`, `daily`, `syslog`, `errorlog`, `slack`;
- `papertrail` — использует библиотеку Monolog для отсылки записей в веб-службу Papertrail (**https://www.solarwinds.com/papertrail/**). Параметры для подключения к службе указываются в настройке `handler_with`, хранящей ассоциативный массив с элементами:
  - `host` — интернет-адрес хоста веб-службы. Значение берется из локальной настройки `PAPERTRAIL_URL`, изначально отсутствующей в файле .env;
  - `port` — номер TCP-порта, через который осуществляется взаимодействие с веб-службой. Значение берется из локальной настройки `PAPERTRAIL_PORT`, изначально отсутствующей в файле .env;
  - `connectionString` — полный интернет-адрес для взаимодействия с веб-службой. Изначально: строка формата:

    `tls://`*<интернет-адрес хоста>*`:`*<номер TCP-порта>*

- `stderr` — использует библиотеку Monolog для вывода записей в стандартный поток ошибок PHP `php://stderr`. Доступны дополнительные настройки:
  - `formatter` — строковый путь к классу форматировщика записей, входящего в состав библиотеки Monolog. Значение берется из локальной настройки `LOG_STDERR_FORMATTER`, изначально отсутствующей в файле .env. Список доступных форматировщиков и описание их использования приведены в документации по библиотеке;
  - `with` — хранит ассоциативный массив с элементом `stream`, в котором записано строковое обозначение стандартного потока PHP (изначально: `php://stderr`, т. е. поток ошибок);
- `null` — никуда не заносит записи. Также использует библиотеку Monolog;
- `emergency` — выводит только записи об ошибках, возникающих в службах журналирования, и ошибках в настройках каналов журналирования. Использует службу `single` и записывает журнал в файл storage\logs\laravel.log;

❒ `default` — имя канала, используемого по умолчанию, если при занесении записи канал не был указан явно. Значение берется из локальной настройки `LOG_CHANNEL`, имеющей изначальное значение `stack`. По умолчанию: также `stack`;

❒ `deprecations` — имя канала, используемого для занесения записей об использовании в коде каких-либо устаревших или нерекомендованных программных инструментов. Значение берется из локальной настройки `LOG_DEPRECATIONS_CHANNEL`, имеющей изначальное значение `null`. По умолчанию: также `null`.

34.1.2. Занесение записей в журнал

Чтобы занести запись с заданными *уровнем* и *текстом* в журнал через канал по умолчанию, следует вызвать у фасада `Illuminate\Support\Facades\Log` метод `log()`:

```
log(<уровень записи>, <текст записи>[, <дополнительная информация>=[]])
```

Уровень записи задается в виде строки. Пример:

```
use Illuminate\Support\Facades\Log;
. . .
Log::log('notice', 'Объявление удалено');
```

Можно указать произвольную *дополнительную информацию*, оформив ее в виде ассоциативного массива. Эта *информация* будет отформатирована и занесена в журнал вместе с записью. Пример:

```
use Psr\Log\LogLevel;
Log::log(LogLevel::NOTICE, 'Объявление удалено',
                           ['bb' => $bb, 'user' => Auth::user()]);
```

Вместо обращения к фасаду `Log` для получения объекта канала журналирования по умолчанию можно вызвать функцию-хелпер `logger()` без параметров — это немного сократит код:

```
logger()->log('notice', 'Объявление удалено',
                        ['bb' => $bb, 'user' => Auth::user()]);
```

Также можно использовать более специализированные методы: `debug()`, `info()`, `notice()`, `warning()`, `error()`, `critical()`, `alert()` и `emergency()`, заносящие в журнал записи соответствующего уровня. Все эти методы имеют сходный формат вызова:

<метод>(*<текст записи>*[, *<дополнительная информация>*=[]])

Примеры:

```
Log::notice('Объявление удалено', ['bb' => $bb, 'user' => Auth::user()]);

logger()->debug('Выполнено действие index() контроллера MainController');
```

Вместо метода `info()` можно использовать функцию `info()`, а вместо метода `debug()` — функцию `logger()`, вызванную с параметрами. Обе функции имеют тот же формат вызова, что и соответствующие им методы. Пример:

```
logger('Выполнено действие index() контроллера MainController');
```

При обращении непосредственно к фасаду `Log` записи будут отправлены по каналу журналирования, указанному в настройках в качестве канала по умолчанию. Если нужно отправить запись по другому каналу, следует сначала вызвать у фасада `Log` один из описанных далее методов, а у возвращенного ими результата вызвать метод, отправляющий запись. Вот два метода, указывающих каналы:

- `channel(<имя канала>)` — выбирает для отправки записи канал журналирования с указанным *именем*:

  ```
  Log::channel('syslog')->critical('Сбой веб-сервера!');
  ```

- `stack(<массив с именами каналов>)` — выбирает для отправки записи каналы журналирования с приведенными в *массиве* именами:

  ```
  Log::stack(['syslog', 'slack'])->info('Создан новый пользователь');
  ```

34.1.2.1. Разделяемая дополнительная информация

Если в журнал в составе разных записей должна заноситься одна и та же дополнительная информация, удобнее указать эту информацию где-либо в одном месте кода, а не записывать ее в каждом вызове метода, заносящего запись.

Если одинаковая информация должна заносится во все записи журнала, создаваемые во время обработки текущего клиентского запроса, следует указать эту информацию в каком-либо посреднике — уже существующем или созданным специально для этой цели (о посредниках рассказывалось в *главе 21*). Для указания дополнительной информации, представляемой ассоциативным массивом, применяется метод `withContext(<ассоциативный массив>)`, вызываемый у фасада `Log`. Пример посредника `App\Http\Middleware\SharedAddInfo`, заносящего в состав дополнительной информации данные, которые были получены от пользователя, приведен в листинге 34.1.

Листинг 34.1. Код посредника `App\Http\Middleware\SharedAddInfo`

```
namespace App\Http\Middleware;
use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
```

```
class SharedAddInfo {
    public function handle(Request $request, Closure $next) {
        Log::withContext(['userdata' => $request->input()]);
        return $next($request);
    }
}
```

34.1.3. Событие, генерируемое при занесении записи в журнал

Сразу после занесения любой записи в журнал генерируется событие `Illuminate\Log\Events\MessageLogged`. Оно содержит следующие свойства:

- `level` — уровень записи в виде строкового обозначения;
- `message` — текст записи;
- `context` — дополнительная информация в виде ассоциативного массива.

34.2. Дополнительные средства отладки

Laravel предоставляет дополнительные средства отладки, позволяющие просмотреть структуру любого массива или объекта, а также SQL-код, генерируемый и отсылаемый СУБД построителем запросов.

Чтобы вывести непосредственно на веб-страницу значение какой-либо переменной, элемента массива, свойства объекта, результата, возвращаемого функцией или методом, нужно использовать одну из двух следующих функций:

- `dd(<`*значение*`>)` — выводит указанное *значение* на страницу (рис. 34.1) и прерывает исполнение кода:

  ```
  dd($rubric);
  ```

  Если выводимое значение является объектом, отображаются все его свойства. Если значение какого-либо свойства является объектом или массивом, правее его имени выводится треугольная стрелка, щелкнув на которой можно просмотреть все свойства этого объекта или элементы этого массива (так, на рис. 34.1 «развернуто» значение свойства `attributes`, представляющее собой массив).

  Поскольку функция `dd()` немедленно прерывает выполнение кода, на веб-странице будут присутствовать только сведения, выведенные этой функцией;

- `dump(<`*значение*`>)` — то же самое, что и `dd()`, только не прерывает исполнение кода. Таким образом, на странице окажутся и сведения, выведенные этой функцией, и содержание страницы, сгенерированное шаблоном.

Для просмотра SQL-кода, сгенерированного построителем запросов, и значений вставляемых в SQL-запрос параметров, предназначены следующие методы, поддерживаемые построителем запросов:

- `dd()` — выводит сначала код SQL-запроса, а потом — массив со значениями вставляемых в запрос параметров (рис. 34.2), после чего прерывает исполнение кода:

  ```
  Bb::where('price', '>', 10000)->orderBy('title')->offset(0)->limit(5)->dd();
  ```

- `dump()` — то же самое, что и `dd()`, только не прерывает исполнение кода.

```
^ App\Models\Rubric {#1418 ▼
  #connection: "sqlite"
  #table: "rubrics"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:5 [▼
    "id" => 8
    "name" => "Дачи"
    "parent_id" => 1
    "created_at" => "2022-05-09 08:25:15"
    "updated_at" => "2022-05-09 08:25:15"
  ]
  #original: array:5 [▶]
  #changes: []
  #casts: []
  #classCastCache: []
  #attributeCastCache: []
  #dates: []
  #dateFormat: null
  #appends: array:1 [▶]
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  #hidden: []
  #visible: []
  #fillable: array:2 [▶]
  #guarded: array:1 [▶]
}
```

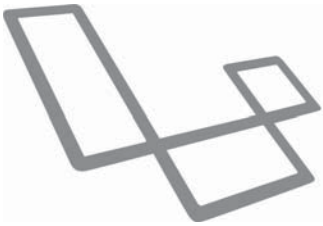
Рис. 34.1. Объект записи, выведенный функцией `dd()`

```
"select * from "bbs" where "price" > ? order by "title" asc limit 5 offset 0"

array:1 [▼
  0 => 10000
]
```

Рис. 34.2. SQL-код и параметры запроса, выведенные методом `dd()`

ГЛАВА 35



Публикация веб-сайта

35.1. Подготовка веб-сайта к публикации

35.1.1. Удаление ненужного кода и данных

Самый первый этап подготовки сайта к публикации включает в себя:

- удаление ненужного кода (неиспользуемых модулей, переменных, свойств и методов);
- удаление кода, используемого только при разработке и отладке (выражений, выводящих на страницы отладочные данные, «заглушек» и пр.);
- удаление ненужных данных (неиспользуемых таблиц стилей, графических изображений, статичных веб-страниц, журналов и пр.).

Для уменьшения объема сайта также можно удалить всевозможные временные данные:

- устаревшие электронные жетоны сброса паролей — набором команды:

```
php artisan auth:clear-resets
```

- миниатюры, созданные на основе выгруженных пользователями изображений (если используется библиотека bkwld/croppa), — командой:

```
php artisan croppa:purge
```

- проваленные отложенные задания — командой:

```
php artisan queue:flush
```

- выполненные пакеты отложенных заданий — командой:

```
php artisan queue:prune-batches
```

- ошибочные пакеты отложенных заданий — командой:

```
php artisan queue:prune-failed
```

- содержимое кеша — командой:

```
php artisan cache:clear [<имя службы кеша>]
```

35.1.2. Настройка под платформу публикации

Далее следует указать настройки, характерные для платформы, на которой будет опубликован сайт. Бóльшая часть этих настроек записывается в файле .env, однако, возможно, придется внести правки и в модули, хранящиеся в папке config.

Скорее всего, придется изменить настройки баз данных (включая Redis), хранилища выгруженных файлов, электронной почты, службы очередей, сессий, кеширования и журналирования. Также, возможно, потребуется исправить настройки доступа к сторонним веб-служб (наподобие Mailgun, Amazon S3 или Slack).

35.1.3. Переключение в режим эксплуатации

Обязательно следует переключить сайт в режим эксплуатации, для чего нужно:

- локальной настройке `APP_ENV` или рабочей настройке `app.env` — дать значение `'production'`.

  В результате ключевые подсистемы сайта будут конфигурироваться более оптимальным для эксплуатации образом;

- локальной настройке `APP_DEBUG` или рабочей настройке `app.debug` — дать значение `false`.

  После этого при возникновении ошибки в коде сайта будут выдаваться страницы с более лаконичными сообщениями вместо подробных, генерируемых при работе в режиме разработки. Благодаря этому злоумышленники не смогут получить сведения о работе сайта и использовать их для взлома;

- локальной настройке `APP_URL` или рабочей настройке `app.url` — присвоить интернет-адрес хоста, на котором будет работать сайт.

  Тот же интернет-адрес нужно занести в настройки, задающие параметры дополнительных библиотек (например, в настройку `services.vkontakte.redirect`, задающую интернет-адрес для перенаправления после успешного входа посредством «ВКонтакте», подробности — в *разд. 19.2*).

35.1.4. Задание списка доверенных прокси-серверов

В последнее время часто используются прокси-серверы, защищающие сайты от сетевых атак или исполняющие роль балансировщиков нагрузки, которые принимают «извне» запросы по протоколу HTTPS и передают их обслуживаемому сайту уже по протоколу HTTP. В таком случае фреймворк, не зная, что клиенты для доступа к сайту используют протокол HTTPS, будет генерировать интернет-адреса для гиперссылок с применением протокола HTTP вместо HTTPS, что может привести к неработоспособности сайта (о генерировании интернет-адресов рассказывалось в *разд. 9.4*).

Решить эту проблему можно, пометив используемые прокси-серверы как доверенные. Для этого достаточно:

1. Открыть корневой класс маршрутизатора `App\Http\Kernel` и проверить, присутствует ли в массиве из свойства `middleware` посредник `App\Http\Middleware\TrustProxies` (т. е. связан ли он со всеми маршрутами). Этот посредник задает список доверенных прокси-серверов.

2. Открыть посредник `App\Http\Middleware\TrustProxies` и присвоить защищенному свойству `proxies` массив с интернет-адресами прокси-серверов, которые нужно сделать доверенными:

```
class TrustProxies extends Middleware {
    protected $proxies = ['192.168.1.1', '192.168.1.2'];
    . . .
}
```

   Если используется облачная служба балансировки нагрузки (например, Amazon AWS), выяснить интернет-адреса отдельных прокси-серверов невозможно. В таком случае следует пометить как доверенные все прокси-серверы, присвоив свойству `proxies` посредника строку `'*'`.

3. При необходимости присвоить защищенному свойству `headers` того же посредника заголовок, вставляемый прокси-серверами в запросы, перенаправляемые сайту. Обнаружив в полученном запросе этот заголовок, фреймворк поймет, что запрос пришел от прокси-сервера.

   Свойству `headers` можно присвоить произвольную комбинацию констант класса `Illuminate\Http\Request`: `HEADER_FORWARDED`, `HEADER_X_FORWARDED_FOR`, `HEADER_X_FORWARDED_HOST`, `HEADER_X_FORWARDED_PROTO`, `HEADER_X_FORWARDED_PORT`, `HEADER_X_FORWARDED_PREFIX`, `HEADER_X_FORWARDED_AWS_ELB` или `HEADER_X_FORWARDED_TRAEFIK`. Пример:

```
class TrustProxies extends Middleware {
    . . .
    protected $headers = Request::HEADER_X_FORWARDED_HOST |
                         Request::HEADER_X_FORWARDED_AWS_ELB;
}
```

   Изначально в этом свойстве указана комбинация из заголовков: `HEADER_X_FORWARDED_FOR`, `HEADER_X_FORWARDED_HOST`, `HEADER_X_FORWARDED_PROTO`, `HEADER_X_FORWARDED_PORT` и `HEADER_X_FORWARDED_AWS_ELB`.

35.1.5. Задание списка доверенных хостов

Внутрикорпоративные сайты почти всегда делаются доступными только из корпоративной сети. Это можно реализовать как посредством прокси-сервера, блокирующего доступ к сайту извне, так и на уровне самого сайта — занеся интернет-адреса, с которых должен быть доступен сайт, в список доверенных хостов.

Для этого следует:

1. Открыть корневой класс маршрутизатора `App\Http\Kernel` и в массиве из свойства `middleware` найти и раскомментировать посредник `App\Http\Middleware\TrustHosts` (после чего он окажется связанным со всеми маршрутами). Этот посредник задает список доверенных хостов и реализует блокировку сайта при попытке доступа к нему извне.
2. Открыть посредник `App\Http\Middleware\TrustHosts` и записать в общедоступный метод `hosts()` код, возвращающий массив с интернет-адресами доверенных хостов. В этом массиве могут присутствовать как собственно интернет-адреса, так и регулярные выражения, задающие шаблоны интернет-адресов. Пример:

```
class TrustHosts extends Middleware {
    public function hosts() {
        return ['^(.+\.)?bbcorp.ru$', 'friendlycorp.ru',
                                      'boss.friendlycorp.ru'];
    }
}
```

В возвращаемый массив можно включить вызов метода `allSubdomainsOfApplicationUrl()` того же посредника, который возвращает регулярное выражение, совпадающее с доменом текущего сайта и всеми его поддоменами:

```
public function hosts() {
    return [ ... , $this->allSubdomainsOfApplicationUrl()];
}
```

35.1.6. Компиляция шаблонов

Перед использованием каждого шаблона Laravel компилирует его в PHP-код и сохраняет в файле, имя которого представляет собой хеш, вычисленный на основе пути к файлу с исходным кодом шаблона. Если исходный шаблон будет изменен, фреймворк перекомпилирует его.

Откомпилированные шаблоны по умолчанию сохраняются в папке storage\framework\views. Можно указать сохранение их в другой папке, записав путь к ней в рабочей настройке `views.compiled`.

Компиляция шаблонов, равно как и проверка, был ли исходный шаблон изменен после последней компиляции, отнимает системные ресурсы. Поэтому перед публикацией сайта рекомендуется принудительно откомпилировать все шаблоны (в документации по фреймворку этот процесс почему-то называют кешированием).

Компиляцию всех шаблонов выполняет команда:

```
php artisan view:cache
```

При необходимости можно удалить все откомпилированные шаблоны, подав следующую команду:

```
php artisan view:clear
```

35.1.7. Кеширование маршрутов

Для ускорения обработки списков маршрутов перед публикацией сайта рекомендуется выполнить их кеширование, при котором все маршруты преобразуются в более оптимальный формат и сохраняются в файле bootstrap\cache\routes-v7.php.

Кеширование маршрутов выполняет команда:

```
php artisan route:cache
```

Кешированию подвергаются также все дополнительные модули с маршрутами, созданные разработчиком сайта (как создавать дополнительные списки маршрутов, было рассказано в *разд. 8.9*).

К сожалению, после добавления нового маршрута, изменения или удаления существующего Laravel не выполняет повторное кеширование маршрутов. Их придется перекешировать вручную, набрав приведенную только что команду.

Удалить модуль с кешированными маршрутами можно набором команды:

```
php artisan route:clear
```

35.1.8. Кеширование настроек

Для ускорения работы сайта перед его публикацией рекомендуется свести все модули с настройками, хранящиеся в папке config, в один, который будет обрабатываться быстрее. Этот процесс называется *кешированием настроек*. Модуль с кешированными настройками сохраняется в файле bootstrap\cache\config.php.

Кеширование настроек выполняется вызовом команды:

```
php artisan config:cache
```

Кешированию подвергаются все модули, которые находятся в папке config, включая созданные разработчиком сайта.

Следует учесть, что после правки настроек сайта они не кешируются повторно автоматически. Их перекеширование придется провести вручную, набрав приведенную только что команду.

Удалить модуль с кешированными настройками можно набором команды:

```
php artisan config:clear
```

35.1.9. Кеширование обработчиков событий

Если на сайте используется обработка событий, имеет смысл выполнить кеширование слушателей событий, сведя их в компактный список. В результате Laravel при возникновении события быстро найдет нужный слушатель, а не будет просматривать в его поисках папку app\Listeners. Модуль со списком слушателей событий, полученным в результате кеширования, хранится в файле bootstrap\cache\events.php.

Кеширование слушателей событий производится командой:

```
php artisan event:cache
```

К сожалению, подписчики кешированию не подвергаются, поэтому при программировании высоконагруженных сайтов их лучше не использовать.

Опять же, при добавлении и удалении слушателей событий их список не обновляется автоматически. Перекеширование слушателей нужно выполнить вручную — набором приведенной только что команды.

Удалить список слушателей, полученный в результате кеширования, можно командой:

```
php artisan event:clear
```

35.1.10. Приведение таблиц стилей и веб-сценариев к виду, оптимальному для публикации

Если при разработке сайта использовался пакет Laravel Mix, обеспечивающий работу с таблицами стилей и веб-сценариями, следует привести их к виду, наиболее подходящему для публикации. При этом Laravel Mix удалит из файлов ненужные пробелы и разрывы строк, по возможности сократит имена переменных и функций, что существенно уменьшит объем файлов и ускорит их загрузку.

Для преобразования таблиц стилей и веб-сценариев к оптимальному для публикации виду следует набрать команду:

```
npm run production
```

35.2. Перенос веб-сайта на платформу для публикации

Скорее всего, сайт будет опубликован на другой платформе. Следовательно, его нужно перенести туда. Для этого необходимо:

1. Скопировать каким-либо образом (перенести на флешке, переписать по сети и т. д.) на целевую платформу все содержимое папки проекта, за исключением следующих папок:

   - vendor — хранит сам фреймворк и все необходимые ему для работы дополнительные библиотеки, написанные на PHP.

     Эти библиотеки могут быть без труда установлены на целевой платформе, поэтому переносить их вместе с сайтом не нужно;

   - node_modules — хранит библиотеки и программы, написанные на JavaScript и работающие под управлением Node.js.

     Эти программы и библиотеки нужны исключительно для разработки и при эксплуатации сайта лишь будут занимать место на диске;

   - tests — хранит тесты, опять же, используемые лишь при разработке.

   Также не следует копировать файлы: phpunit.xml, .editorconfig, .env.example, README.md,.styleci.yml и webpack.mix.js, которые нужны лишь при разработке.

   Следующие действия выполняются на целевой платформе:

2. В командной строке перейти в папку, в которую скопирован сайт.
3. Установить библиотеки, написанные на PHP, включая сам фреймворк, набрав команду:

   ```
   composer install --optimize-autoloader --no-dev
   ```

   Командный ключ `--no-dev` указывает установить лишь библиотеки, необходимые для эксплуатации, а ключ `--optimize-autoloader` — оптимизировать PHP-модуль автозагрузки для повышения производительности.

4. Если сайт позволяет пользователям выгружать файлы — создать символическую ссылку на папку storage\app\public, подав команду:

```
php artisan storage:link [--relative]
```

5. Создать базы данных, используемые сайтом, либо выполнив миграции, либо на основе ранее созданного дампа (см. *разд. 4.1.8*).

35.3. Настройка веб-сервера и запуск сторонних программ

В настройках веб-сервера, который будет обслуживать опубликованный сайт, в качестве корневой папки сайта следует указать папку public. Вот пример кода, конфигурирующего веб-сервер Apache HTTP Server (при условии, что сайт хранится в папке D:\published_sites\bboard):

```
DocumentRoot "d:/published_sites/bboard/public"
<Directory "d:/published_sites/bboard/public">
    . . .
</Directory>
```

В папке public уже имеется файл .htaccess, должным образом настраивающий Apache HTTP Server для работы с сайтом. Код аналогичного конфигурирующего файла для веб-сервера Nginx можно найти по интернет-адресу: **https://laravel.com/docs/9.x/deployment#nginx**.

Если сайт обрабатывает отложенные задания, нужно запустить их обработчик с помощью команды:

```
php artisan queue:work
```

Если сайт использует планировщик заданий, последний следует запустить либо посредством планировщика операционной системы, либо в независимом режиме — командой:

```
php artisan schedule:work [--quiet]
```

Если сайт реализует вещание с применением сервера Laravel Websockets, последний нужно запустить посредством команды:

```
php artisan websockets:serve
```

После чего можно запускать сам веб-сервер.

35.4. Режим обслуживания

Если требуется провести какие-либо технические работы на уже опубликованном сайте (удалить ненужные и мусорные данные, исправить ошибки в коде и пр.), сайт следует закрыть от посетителей, переведя его в *режим обслуживания*.

Перевод сайта в режим обслуживания осуществляется командой:

```
php artisan down [--secret="<секретный жетон>"] ↳
[--redirect=<путь перенаправления>] ↳
```

```
[--refresh=<выжидаемое время>][--retry=<выжидаемое время>] ↳
[--status=<код ошибки>] [--render="<путь к шаблону>::<код ошибки>"]
```

После этого при попытке открыть сайт будет выводиться стандартная страница сообщения об ошибке 503 с надписью «Service Unavailable» (или «Сервис недоступен», если была выполнена локализация сайта посредством библиотеки Laravel Lang, описанной в *разд. 28.4*).

Поддерживаются следующие командные ключи:

- `--secret` — задает *секретный жетон*, позволяющий все же перейти на сайт, набрав интернет-адрес формата:

  <целевой интернет-адрес>/<заданный секретный жетон>

  Например, указав жетон `123456789`:

  ```
  php artisan down --secret="123456789"
  ```

  и набрав интернет-адрес **http://bboard.ru/123456789/**, можно перейти на сайт, опубликованный по интернет-адресу: **http://bboard.ru/**;

- `--redirect` — задает *путь*, по которому будет выполняться предварительное перенаправление перед выдачей страницы с сообщением о недоступности сайта. Например, так можно указать выполнять предварительное перенаправление в «корень» сайта:

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

- `--refresh` — указывает *время* в секундах, спустя которое веб-обозреватель выполнит самостоятельную попытку вновь попасть на сайт, обновив страницу. Это *время* будет пересылаться клиенту в составе серверного ответа с сообщением о недоступности сайта в заголовке `Refresh`. Пример указания повторно попытаться попасть на сайт спустя минуту:

  ```
  php artisan down --refresh=60
  ```

- `--retry` — то же самое, что и `--refresh`, только пересылает заданное *время* клиенту в заголовке `Retry-After` (который многие современные веб-обозреватели игнорируют);

- `--status` — задает *код ошибки*, который будет выводиться на странице с сообщением о недоступности сайта, вместо 503.

  Когда сайт, находящийся в режиме обслуживания, обрабатывает клиентский запрос, задействуется довольно много подсистем фреймворка. Если код одной из этих подсистем в текущий момент переписывается, может возникнуть ошибка, и сайт полностью перестанет работать;

- `--render` — указывает сайту сразу после получения клиентского запроса вывести страницу на основе шаблона с заданным *путем* и отобразить на ней указанный *код ошибки*. При этом задействуется минимум подсистем фреймворка и вероятность возникновения ошибки существенно снижается. Пример:

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

Для перевода сайта из режима обслуживания в рабочий режим необходимо набрать команду:

```
php artisan up
```

Изначально сайт, находящийся в режиме обслуживания, недоступен полностью — ни один посетитель не сможет зайти ни на одну его страницу (исключение составят лишь счастливые обладатели секретного жетона, который, впрочем, может быть и не задан). Однако часть страниц сайта все же можно сделать доступными для посетителей. Для этого следует:

1. Открыть корневой класс маршрутизатора `App\Http\Kernel` и проверить, присутствует ли в массиве из свойства `middleware` посредник `App\Http\Middleware\PreventRequestsDuringMaintenance` (т. е. связан ли он со всеми маршрутами). Этот посредник задает список путей страниц, которые должны быть доступными всегда.
2. Открыть посредник `App\Http\Middleware\PreventRequestsDuringMaintenance` и присвоить защищенному свойству `except` массив с путями к страницам, которые нужно сделать доступными для посетителей. В составе путей можно использовать литералы `*`, обозначающие произвольное количество любых символов. Пример:

```
class PreventRequestsDuringMaintenance extends Middleware {
    protected $except = ['/news/*', '/policy', '/about'];
}
```

Заключение

Вот и закончилась книга о Laravel, наиболее популярном в настоящее время веб-фреймворке, написанном на языке PHP. Она получилась довольно толстой, поскольку автор постарался как можно подробнее описать в ней все часто применяемые на практике программные инструменты, предлагаемые этим замечательным фреймворком, привести примеры их практического применения и рассказать о нескольких полезных дополнительных библиотеках. Для чего ему пришлось прочитать (и не раз) официальную документацию, поискать дополнительные источники в Интернете, изучить программный код Laravel и, безусловно, поэкспериментировать.

Laravel — фреймворк воистину всеобъемлющий. Иногда возникает впечатление, что в нем есть буквально все, что нужно типичному веб-разработчику прямо сейчас, и многое из того, что может пригодиться потом. Неудивительно, что все это богатство не поместилось в одну книгу, и кое-что осталось, что называется, за кадром, вследствие ограниченного объема книги:

- класс `Carbon`, предназначенный для представления временны́х отметок и являющийся производным от встроенного в PHP класса `DateTime`;
- специализированные инструменты для работы с базами данных Redis;
- встроенный HTTP-клиент (может пригодиться, если сайт загружает и обрабатывает информацию с других сайтов);
- подсистема автоматического тестирования (которую автор не считает сколь-нибудь полезной);
- разработка своих собственных служб очередей, сессий, кеширования, журналирования и др.;
- разработка дополнительных библиотек для Laravel;
- отладочная панель Laravel Telescope;
- подсистема электронной коммерции Laravel Cashier;
- служба аутентификации для бэкендов Laravel Passport;
- административные панели, предоставляющие готовые инструменты для работы с внутренними данными сайта (их существует довольно много, как бесплатных, так и платных);

- множество других дополнительных библиотек, могущих оказаться полезными в ряде случаев.

Пожалуй, за кадром осталось даже слишком много… Но что поделаешь, став в 2015 году номером один среди веб-фреймворков и с тех пор не уступив этого места никому, Laravel за прошедшие годы обзавелся внушительным набором как встроенных функциональных возможностей, так и расширений, написанных сторонними разработчиками. Чтобы рассказать обо всем этом, придется писать многотомник…

К счастью, необходимую документацию можно найти в Интернете. Интернет-адреса некоторых источников дополнительной информации по тем или иным темам были приведены непосредственно в тексте глав книги. Еще несколько сайтов, полезных Laravel-программисту, приведены в табл. З.1.

Таблица З.1. *Интернет-ресурсы, посвященные Laravel*

Интернет-адрес	Описание
https://laravel.com/	Официальный сайт Laravel. Помимо документации по самому фреймворку здесь также имеются описания дополнительных библиотек, написанных самими разработчиками Laravel: Cashier, Passport, Telescope и др.
https://laravel-news.com/	Официальный блог Laravel. Публикуются новости и полезные статьи
https://packalyst.com/	Хранилище дополнительных библиотек, расширяющих возможности Laravel
https://laravel.su/	Сайт русского сообщества Laravel. Имеется документация, переведенная на русский язык, правда, по версии 8.0
https://vk.com/laravel_rus	Наиболее крупное сообщество «ВКонтакте», посвященное Laravel
https://getcomposer.org/	Официальный сайт утилиты Composer

На этом автор книги прощается с вами, уважаемые читатели. Успехов вам в сайтостроении — занятии нелегком, но чрезвычайно увлекательном!

Владимир Дронов

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив, сопровождающий книгу, выложен на сервер издательства «БХВ» по интернет-адресу: **https://zip.bhv.ru/9785977517256.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Содержимое архива описано в табл. П.1.

Таблица П.1. *Содержимое электронного архива*

Папка, файл	**Описание**
bboard	Папка с исходным кодом учебного веб-сайта доски объявлений, разрабатываемого на протяжении *глав 1* и *2* книги на PHP и Laravel
readme.txt	Файл с описанием архива и инструкцией по развертыванию учебного веб-сайта

Предметный указатель

_

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

А

Б

В

Г

Д

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я