

O'REILLY®

2-е издание

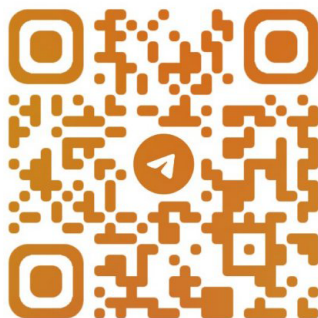
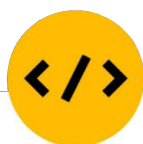
Laravel

Полное руководство



 ПИТЕР®

Мэтт Стаффер



@CODELIBRARY_IT

SECOND EDITION

Laravel: Up & Running

A Framework for Building Modern PHP Apps

Matt Stauffer

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY

Мэтт Стаффер

Laravel

Полное руководство

2-е издание



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

Мэтт Стаффер

Laravel. Полное руководство. 2-е издание

Серия «Бестселлеры O'Reilly»

Перевел с английского *Владимир Сауль*

Заведующая редакцией
Руководитель проекта
Ведущий редактор
Художественный редактор
Литературные редакторы
Корректоры
Верстка

*Ю. Сергиенко
С. Давид
Н. Гринчик
Н. Васильева
В. Байдук, А. Дубейко
О. Андриевич, Е. Павлович
Г. Блинов*

ББК 32.988.02-018

УДК 004.738.5

Стаффер Мэтт

C78 **Laravel. Полное руководство. 2-е изд. — СПб.: Питер, 2020. — 512 с.: ил. — (Серия «Бестселлеры O'Reilly»).**

ISBN 978-5-4461-1396-5

Что отличает Laravel от других РНР-фреймворков? Скорость и простота. Стремительная разработка приложений, обширная экосистема и набор инструментов Laravel позволяют быстро создавать сайты и приложения, отличающиеся чистым и удобочитаемым кодом.

Мэтт Стаффер, известный преподаватель и ведущий разработчик, предлагает как общий обзор фреймворка, так и конкретные примеры работы с ним. Опытным РНР-разработчикам книга поможет быстро войти в новую тему, чтобы реализовать проект на Laravel. В издании также раскрыты темы Laravel Dusk и Horizon, собрана информация о ресурсах сообщества и других пакетах, не входящих в ядро Laravel.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1492041214 англ. Authorized Russian translation of the English edition of Laravel: Up and Running, 2nd Edition. ISBN 9781492041214 © 2019 Matt Stauffer
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1396-5 © Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Бестселлеры O'Reilly», 2020

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.11.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Предисловие	20
От издательства	25
Глава 1. Зачем использовать Laravel	26
Глава 2. Настройка среды разработки для использования Laravel.....	36
Глава 3. Маршрутизация и контроллеры.....	47
Глава 4. Движок шаблонов Blade.....	86
Глава 5. Базы данных и Eloquent	108
Глава 6. Компоненты для клиентской части.....	177
Глава 7. Получение и обработка пользовательских данных	198
Глава 8. Интерфейсы Artisan и Tinker	216
Глава 9. Аутентификация и авторизация пользователей.....	234
Глава 10. Запросы, ответы и middleware	267
Глава 11. Контейнер	293
Глава 12. Тестирование	308
Глава 13. Создание API	350
Глава 14. Сохранение и извлечение данных	390
Глава 15. Почта и уведомления	418
Глава 16. Очереди, задания, события, трансляция и планировщик	439
Глава 17. Хелперы и коллекции	477
Глава 18. Экосистема инструментов Laravel	494
Глоссарий.....	501
Об авторе.....	511
Об обложке.....	512

Оглавление

Предисловие	20
О чем эта книга.....	20
Для кого предназначена книга.....	21
Структура издания.....	21
О втором издании.....	21
Условные обозначения.....	22
Благодарности	23
Первое издание.....	23
Второе издание	24
От издательства	25
Глава 1. Зачем использовать Laravel	26
Для чего нужен фреймворк.....	26
«Все своими руками».....	27
Согласованность и гибкость	27
Краткий экскурс в историю веб- и PHP-фреймворков	27
Ruby on Rails.....	28
Бум PHP-фреймворков.....	28
Преимущества и недостатки CodeIgniter	28
Laravel 1, 2 и 3.....	29
Laravel 4.....	29
Laravel 5.....	29
Чем уникален Laravel.....	30
Философия Laravel	30
Как Laravel делает разработчиков счастливее	31
Сообщество Laravel.....	32
Как работает Laravel.....	32
Почему стоит выбрать Laravel	35

Глава 2. Настройка среды разработки для использования Laravel	36
Системные требования	36
Composer	37
Локальные среды разработки	37
Laravel Valet	37
Laravel Homestead	38
Создание нового проекта Laravel	38
Установка Laravel с помощью установщика Laravel	39
Установка Laravel с помощью функции create-project менеджера пакетов Composer	39
Lambo: улучшенный вариант команды laravel new	39
Структура каталогов Laravel	40
Каталоги	40
Отдельные файлы	41
Конфигурация	42
Завершение настройки	45
Тестирование	45
Резюме	46
Глава 3. Маршрутизация и контроллеры	47
Краткое введение в MVC, команды HTTP и REST	47
Что такое MVC	47
HTTP-команды	48
Что такое REST	49
Определения маршрутов	50
Команды маршрутов	51
Обработка маршрутов	52
Параметры маршрутов	53
Имена маршрутов	54
Группы маршрутов	56
Middleware	57
Префиксы путей	59
Запасные маршруты	60
Поддоменная маршрутизация	60
Префиксы пространства имен	61
Префиксы имен	61
Подписанные маршруты	62
Подписание маршрута	62
Изменение маршрутов для разрешения подписанных ссылок	63

Представления (views)	64
Прямой возврат простых маршрутов с помощью метода <code>Route::view()</code>	65
Общий доступ представлений к переменным с использованием компоновщиков представлений	65
Контроллеры (controllers)	65
Получение ввода пользователя	68
Внедрение зависимостей в контроллеры	69
Контроллеры ресурсов	71
Контроллеры ресурсов API	72
Контроллеры одиночного действия	72
Привязка модели маршрута	73
Неявная привязка модели маршрута	73
Пользовательская привязка модели маршрута	74
Кэширование маршрутов	75
Подмена метода формы	75
HTTP-команды в Laravel	76
Подмена HTTP-метода в HTML-формах	76
Защита CSRF	76
Перенаправления	78
<code>redirect()->to()</code>	79
<code>redirect()->route()</code>	79
<code>redirect()->back()</code>	80
Другие методы перенаправления	80
<code>redirect()->with()</code>	81
Отмена запроса	82
Пользовательские ответы	83
<code>response()->make()</code>	83
<code>response()->json()</code> и <code>->jsonp()</code>	83
<code>response()->download()</code> , <code>->streamDownload()</code> и <code>->file()</code>	83
Тестирование	84
Резюме	85
Глава 4. Движок шаблонов Blade	86
Отображение данных	87
Управляющие структуры	88
Условные конструкции	88
Циклы	89
Наследование шаблонов	90
Определение разделов страницы с помощью директив <code>@section/@show</code> и <code>@yield</code>	90
Включение составляющих представления	93

Использование стеков	94
Использование компонентов и слотов	96
Компоновщики представлений и внедрение сервисов	98
Привязка данных к представлениям с использованием компоновщиков представлений	98
Внедрение сервиса Blade	101
Пользовательские директивы Blade	102
Параметры пользовательских директив Blade	104
Пример: применение пользовательских директив Blade для многоклиентских приложений	104
Упрощенные пользовательские директивы для операторов if	105
Тестирование	106
Резюме	107
Глава 5. Базы данных и Eloquent	108
Конфигурация	108
Подключения базы данных	109
Другие параметры конфигурации базы данных	110
Миграции	110
Определение миграций	111
Запуск миграций	117
Наполнение базы данными	118
Создание сидера	119
Фабрики моделей	120
Генератор запросов	124
Стандартное использование фасада DB	125
Чистый SQL	126
Выстраивание цепочки с генератором запросов	127
Транзакции	135
Введение в Eloquent	136
Создание и определение моделей Eloquent	138
Получение данных с помощью Eloquent	139
Вставки и обновления с помощью Eloquent	141
Удаление с помощью Eloquent	145
Области видимости	147
Настройка взаимодействия полей с аксессорами, мутаторами и приведением атрибутов	150
Коллекции Eloquent	154
Сериализация Eloquent	156
Связи в Eloquent	158
Обновление меток времени родительской записи дочерними записями	170

События Eloquent.....	172
Тестирование	174
Резюме.....	176
Глава 6. Компоненты для клиентской части.....	177
Laravel Mix	177
Структура каталога Mix	179
Запуск Mix.....	179
Что предоставляет Mix	180
Предустановки клиентской части и генерация кода аутентификации	186
Предустановки клиентской части.....	187
Генерация кода аутентификации	188
Разбивка на страницы.....	188
Разбивка на страницы результатов из базы данных.....	188
Создание разбивщиков страниц вручную.....	189
Пакеты сообщений.....	190
Строковые хелперы, множественность и локализация.....	192
Строковые хелперы и множественность	192
Локализация	193
Тестирование	196
Тестирование пакетов сообщений и ошибок	196
Перевод и локализация.....	197
Резюме.....	197
Глава 7. Получение и обработка пользовательских данных	198
Внедрение объекта запроса.....	198
\$request->all()	199
\$request->except() и \$request->only().....	199
\$request->has()	200
\$request->input()	200
\$request->method() и ->isMethod()	201
Ввод массива.....	201
Ввод JSON (и \$request->json())	201
Маршрутные данные.....	203
Из Request	203
Из параметров маршрута	203
Загруженные файлы.....	203
Валидация	206
Метод validate() объекта Request.....	206
Ручная валидация.....	208

Объекты пользовательских правил	208
Отображение валидационных сообщений	209
Запросы формы	209
Создание запроса формы	210
Использование запроса формы	211
Модель массового назначения Eloquent	212
Синтаксис {} и {!!	213
Тестирование	213
Резюме	215
Глава 8. Интерфейсы Artisan и Tinker	216
Введение в интерфейс Artisan	216
Основные команды Artisan	217
Параметры	217
Сгруппированные команды	218
Написание пользовательских команд Artisan	220
Пример команды	222
Аргументы и параметры	223
Использование ввода	225
Приглашения	226
Вывод	227
Команды на основе замыканий	229
Вызов команд Artisan в нормальном коде	229
Tinker	230
Сервер дампа Laravel	231
Тестирование	232
Резюме	233
Глава 9. Аутентификация и авторизация пользователей	234
Модель User и миграция	235
Использование глобального хелпера auth() и фасада Auth	238
Контроллеры аутентификации	239
Контроллер RegisterController	239
Контроллер LoginController	240
Контроллер ResetPasswordController	242
Контроллер ForgotPasswordController	242
Контроллер VerificationController	243
Метод Auth::routes()	243
Каркас аутентификации	244
Токен «Запомнить меня»	245

Выполнение вручную аутентификации пользователей	246
Выполнение вручную выхода пользователя из системы	247
auth	248
Верификация адресов электронной почты	249
Blade-директивы для аутентификации.....	249
Гарды.....	250
Указание другого гарда по умолчанию	250
Использование других гардов без изменения базового	251
Добавление нового гарда.....	251
Гарды на основе замыкания запроса	252
Создание собственного провайдера пользователей.....	252
Собственные провайдеры пользователей для нереляционных баз данных	253
События аутентификации.....	253
Система авторизации (список управления доступом) и роли	254
Определение правил авторизации	255
Фасад Gate (и его внедрение).....	256
Ресурсы гейтов.....	256
Authorize.....	257
Авторизация внутри контроллера	257
Проверка с помощью экземпляра класса User	259
Проверки с помощью Blade-директив.....	260
Перехват проверок.....	260
Политики	261
Тестирование	263
Резюме.....	266
Глава 10. Запросы, ответы и middleware	267
Жизненный цикл запроса в Laravel	267
Начальная загрузка приложения	267
Сервис-провайдеры.....	269
Объект Request.....	271
Получение объекта Request в Laravel	271
Получение основной информации о запросе.....	272
Объект Response.....	276
Использование и создание объектов Response в контроллерах	276
Специализированные типы ответов.....	277
Laravel и middleware	283
Вводная информация о middleware	283
Создание собственного middleware	284

Привязка middleware	286
Передача параметров middleware	289
Доверенные прокси-серверы	290
Тестирование	291
Резюме	292
Глава 11. Контейнер	293
Вводная информация о внедрении зависимостей	293
Внедрение зависимостей и Laravel.....	295
Глобальный хелпер app().....	295
Как осуществляется привязка к контейнеру	296
Привязка классов к контейнеру	297
Привязка к замыканию	298
Привязка одиночек, псевдонимов и экземпляров.....	299
Привязка конкретного экземпляра к интерфейсу	300
Контекстная привязка.....	300
Внедрение в конструктор в файлах Laravel	301
Внедрение через метод	302
Фасады и контейнер.....	303
Как работают фасады.....	304
Фасады реального времени	305
Сервис-провайдеры	306
Тестирование	306
Резюме	307
Глава 12. Тестирование	308
Основы тестирования	309
Именованые тестов.....	313
Среда тестирования	314
Трейты тестирования.....	314
RefreshDatabase.....	315
WithoutMiddleware	315
DatabaseMigrations	315
DatabaseTransactions.....	315
Простые модульные тесты	316
Как осуществляется тестирование приложений.....	317
HTTP-тесты.....	318
Тестирование простых страниц с помощью вызова <code>\$this->get()</code> и других HTTP-вызовов.....	318
Тестирование API на базе JSON с помощью вызова <code>\$this->getJson()</code> и других HTTP-вызовов на базе JSON	319

Утверждения в отношении объекта \$response	320
Аутентификация ответов	322
Ряд других настроек HTTP-тестов.....	323
Обработка исключений в тестах приложений.....	323
Тесты базы данных.....	324
Использование фабрик моделей в тестах.....	325
Заполнение начальными данными в тестах	325
Тестирование других систем Laravel.....	325
Подделка событий	326
Подделка фасадов Bus и Queue.....	327
Подделка фасада Mail.....	328
Подделка фасада Notification.....	329
Подделка фасада Storage	330
Имитирование	331
Вводная информация об имитировании	331
Вводная информация о Mockery	331
Подделка других фасадов	334
Тестирование команд Artisan.....	335
Браузерные тесты	336
Выбор инструмента.....	337
Тестирование с использованием Dusk	338
Резюме.....	349
Глава 13. Создание API	350
Базовые сведения о REST-подобных API на базе JSON.....	350
Организация контроллеров и возвращаемые JSON-сообщения	352
Чтение и отправка заголовков	355
Отправка заголовков ответа в Laravel	356
Чтение заголовков запроса в Laravel	356
Разбивка на страницы в Eloquent.....	356
Сортировка и фильтрация	358
Сортировка результатов API.....	359
Фильтрация результатов API.....	360
Преобразование результатов.....	361
Создание собственного преобразователя	362
Вложение связей пользовательских преобразователей	363
Ресурсы API	365
Создание класса ресурса	365
Коллекции ресурсов.....	366
Вложение связей	368

Применение разбивки на страницы к ресурсам API	369
Условное применение атрибутов	370
Другие настройки для ресурсов API	370
Аутентификация API с помощью Laravel Passport	370
Вводная информация о OAuth 2.0	370
Установка пакета Passport	371
API пакета Passport	373
Типы допуска, предлагаемые пакетом Passport	373
Управление клиентами и токенами с помощью API пакета Passport и компонентов Vue	382
Области видимости пакета Passport	384
Развертывание пакета Passport	386
Аутентификация с помощью токенов API	386
Настройка ответов с кодом 404	387
Тестирование	388
Резюме	389
Глава 14. Сохранение и извлечение данных	390
Локальные и облачные файловые менеджеры	390
Настройка доступа к файлам	390
Использование фасада Storage	392
Добавление дополнительных провайдеров из пакета Flysystem	393
Базовые способы загрузки файлов на сервер и манипулирования файлами	393
Простые способы скачивания файлов	395
Сессии	395
Получение доступа к сессии	395
Методы, доступные в экземплярах сессий	396
Флеш-память сессии	398
Кэш	398
Получение доступа к кэшу	399
Методы, доступные в экземплярах кэшей	400
Cookie-файлы	401
Cookie-файлы в Laravel	401
Получение доступа к cookie-файлам	401
Логирование	404
Когда и зачем следует выполнять логирование	405
Внесение записей в логи	405
Каналы логирования	406
Полнотекстовый поиск с использованием Laravel Scout	409
Установка пакета Scout	409
Пометка модели для индексирования	410

Поиск по вашему индексу	410
Очереди и Scout	410
Выполнение операций без индексирования	411
Условное индексирование моделей	411
Запуск индексирования вручную с помощью кода	411
Запуск индексирования вручную с помощью интерфейса командной строки	412
Тестирование	412
Сохранение файлов	412
Сессия	414
Кэш	415
Cookie-файлы	415
Логирование	416
Scout	417
Резюме	417
Глава 15. Почта и уведомления	418
Почта	418
Классическая электронная почта	419
Простейший способ использования отправлений	419
Шаблоны писем	421
Методы, доступные в build()	422
Прикрепленные файлы и встроенные изображения	423
Markdown-отправления	424
Визуализация отправлений в браузере	426
Очереди	426
Локальная разработка	427
Уведомления	428
Определение метода via() для уведомляемых объектов	431
Отправка уведомлений	432
Помещение уведомлений в очередь	432
Предлагаемые по умолчанию типы уведомлений	433
Тестирование	437
Электронная почта	437
Уведомления	438
Резюме	438
Глава 16. Очереди, задания, события, трансляция и планировщик	439
Очереди	439
Зачем нужны очереди	440
Базовая конфигурация очередей	440

Задания в очереди.....	441
Запуск обработчика очередей.....	444
Обработка ошибок.....	445
Управление очередью.....	447
Очереди для поддержки других функций.....	448
Laravel Horizon	448
События	449
Запуск события.....	449
Прослушивание события	451
Трансляция событий посредством веб-сокетов и Laravel Echo	454
Конфигурация и настройка.....	455
Трансляция события.....	455
Получение сообщения.....	458
Продвинутые инструменты трансляции	460
Laravel Echo (сторона JavaScript-кода).....	464
Планировщик.....	469
Доступные типы задач.....	470
Доступные временные интервалы	470
Определение часовых поясов для запланированных задач	472
Блокирование и наложение.....	472
Обработка выходных данных задачи	473
Перехват задач	474
Тестирование	474
Резюме.....	476
Глава 17. Хелперы и коллекции	477
Хелперы	477
Массивы	477
Строки.....	479
Пути приложения	481
URL-адреса	482
Прочее.....	484
Коллекции.....	486
Базовые сведения.....	487
Некоторые методы.....	489
Резюме.....	493
Глава 18. Экосистема инструментов Laravel	494
Инструменты, рассмотренные в книге.....	494
Valet	494
Homestead	494

Установщик Laravel	495
Mix	495
Dusk	495
Passport	495
Horizon	495
Echo	496
Инструменты, не рассмотренные в книге	496
Forge	496
Envoyer	496
Cashier	497
Socialite	498
Nova	498
Spark	498
Lumen	498
Envoy	499
Telescope	499
Другие ресурсы	499
Глоссарий	501
Об авторе	511
Об обложке	512

Эта книга посвящена моей семье.

Тебе, Миа, моя маленькая принцесса и источник радости и энергии.

Тебе, Малакай, мой маленький принц, искатель приключений и эмпат.

Тебе, Терева, мое вдохновение, мой мотиватор, мое ребро.

Предисловие

История моего знакомства с Laravel вполне заурядна: много лет я писал код на PHP и активно изучал потенциал Rails и других современных веб-фреймворков. В Rails меня привлекало прекрасное сочетание исходных настроек и гибкости, мощные возможности системы управления пакетами стандартного кода Ruby Gems, а также наличие активного сообщества программистов.

Я так и не перешел на Rails, чему был безумно рад, когда узнал о Laravel. Он взял лучшее от Rails, не становясь при этом его клоном. Это был инновационный фреймворк с отличной документацией и доброжелательным сообществом.

После этого я начал делиться своим опытом изучения Laravel: вел блог, записывал подкасты и выступал на конференциях. С помощью Laravel я написал десятки приложений в рамках своей основной работы и дополнительных проектов, а также познакомился лично и в режиме онлайн с разработчиками, использующими этот фреймворк. Даже имея богатый арсенал инструментов разработки, я наслаждаюсь, когда набираю в командной строке `laravel new projectName`.

О чем эта книга

Это не первая книга о Laravel и не последняя. Я не стремился объяснить каждую строку кода или шаблон реализации, поскольку не хотел рассказывать о том, что может устареть после обновления Laravel. Я хотел написать книгу, которая предоставляла бы разработчикам обзор и давала конкретные примеры требований для работы в кодовых базах Laravel с использованием любой функции или подсистемы этого фреймворка. Я стремился не просто дублировать документацию, а помочь вам понять основополагающие концепции Laravel.

Laravel — это мощный и гибкий PHP-фреймворк с постоянно растущим сообществом программистов и широкой экосистемой инструментов, что с каждым днем повышает его привлекательность и доступность. Книга предназначена для разработчиков, которые уже знают, как создавать сайты и приложения, и хотят узнать, как это можно эффективно делать с помощью Laravel.

Документация Laravel всесторонняя и качественная. Если вам кажется, что я недостаточно хорошо осветил определенную тему, то рекомендую ознакомиться с ее более подробным описанием в онлайн-документации по адресу <https://laravel.com/docs>.

Надеюсь, в книге вы найдете оптимальный баланс между теорией и практикой в виде примеров конкретного применения, а по прочтении сможете легко написать с помощью Laravel целое приложение с нуля.

Для кого предназначена книга

Книга подойдет для читателя, знающего базовые методы объектно-ориентированного программирования, язык PHP (или по крайней мере общий синтаксис языков C), а также базовые концепции архитектурного паттерна «Модель — Представление — Контроллер» (Model — View — Controller, MVC) и обработки шаблонов. Если вы никогда не создавали сайт, материал книги может оказаться слишком сложным. Но если у вас есть опыт программирования, то не обязательно знать что-то о Laravel — я объясню все, что нужно, начиная с простейшего примера Hello, world!.

Laravel может работать в любой операционной системе, но приведенные здесь примеры команд оболочки bash проще запускать в Linux/macOS. Пользователям Windows будет сложнее выполнять эти команды и в целом применять современные средства разработки на PHP, однако, следуя инструкциям, вы сможете установить Homestead (виртуальную машину Linux) и запускать все необходимые команды.

Структура издания

В этой книге я старался придерживаться хронологического порядка: сначала рассматриваются базовые компоненты, которые вы будете применять в начале создания веб-приложения с помощью Laravel, а затем — менее фундаментальные и реже используемые возможности.

Главы организованы так, чтобы незнакомые с фреймворком пользователи могли эффективно усваивать материал, читая книгу по порядку.

Большинство глав заканчивается двумя разделами: «Тестирование» и «Краткие итоги». В них соответственно показывается, как писать тесты для описанных возможностей, и проводится общий обзор рассмотренного материала.

Книга написана для Laravel 5.8, но охватывает возможности и нововведения синтаксиса предыдущих версий, начиная с Laravel 5.1.

О втором издании

Первое издание книги вышло в ноябре 2016 года и освещало возможности версий Laravel с 5.1 по 5.3. В новом издании дополнительно рассмотрены возможности версий 5.4–5.8 и инструментов Laravel Dusk и Laravel Horizon, а также добавлена глава 18, посвященная ресурсам сообщества и дополнительным пакетам Laravel, которые не были охвачены в первых 17 главах.

Условные обозначения

В этой книге используются следующие типографские обозначения.

Рубленый шрифт

Используется для выделения URL-адресов и адресов электронной почты.

Курсивный шрифт

Применяется для выделения новых терминов и слов, на которых сделан акцент.

Моноширинный шрифт

Используется для записи листингов программ, а также для выделения в тексте таких элементов, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена и расширения файлов.

Полужирный моноширинный шрифт

Предназначен для выделения команд или другого текста, который должен вводиться пользователем без каких-либо изменений.

Курсивный моноширинный шрифт

Применяется для обозначения в коде элементов, которые требуется заменить предоставленными пользователем значениями или значениями, зависящими от контекста.

{Курсивный моноширинный шрифт в фигурных скобках}

Используется для выделения имен файлов или путей к файлам, которые требуется заменить предоставленными пользователем значениями или значениями, зависящими от контекста.



Так обозначается совет или предложение.



Так обозначается общее примечание.



Так обозначается предупреждение.

Б.х Поскольку эта книга охватывает возможности Laravel версий 5.1–5.8, в книге используются метки для обозначения комментариев, относящихся к той или иной конкретной версии. В общем случае такая метка указывает, в какой версии Laravel

была введена рассматриваемая возможность (так, например, метка «5.3» означает, что рассматриваемая возможность доступна только в Laravel 5.3 и в более новых версиях).

Благодарности

Первое издание

Эта книга не увидела бы свет без поддержки моей удивительной жены Теревы и понимания («Папа пишет, дружище!») моего сына Малакая. В силу возраста моя дочь Миа не осознавала этого, но тоже присутствовала при написании книги, и потому я посвящаю книгу своей семье. Работа заставила меня не раз засиживаться допоздна и проводить выходные дни в кофейнях Starbucks, и я очень признателен жене и детям за то, что они поддерживали и вдохновляли меня уже одним своим присутствием.

В ходе написания книги меня также поддерживал и подбадривал весь коллектив компании Tighten, а некоторые коллеги даже редактировали примеры кода (особенно хочется отметить Кита Дамиани) и помогли мне разобраться с особенно сложными из них (например, Адам Уотан, король конвейера коллекций). Дэн Шитц, мой партнер в компании Tighten, великодушно подстраховывал меня, когда приходилось уходить с работы, чтобы поработать над книгой, поддерживал и ободрял; а Дейв Хиккинг, наш операционный менеджер, помог найти место для написания в моем рабочем графике.

Тейлор Отвел заслуживает благодарности и уважения за создание фреймворка Laravel, что обеспечило работой многих людей и улучшило жизнь многих разработчиков. Самой высокой оценки достойны его забота о разработчиках и его усилия по обращению внимания на их нужды и запросы, созданию позитивного и стимулирующего сообщества. Кроме того, я хочу поблагодарить его как доброго друга, способного поддержать и подтолкнуть к новым свершениям. Тейлор, ты — лучший!

Спасибо Джеффри Вэю — одному из лучших онлайн-преподавателей. Он познакомил меня с Laravel и продолжает знакомить с этим фреймворком все новых и новых людей. Он также прекрасный человек, и я очень рад, что могу называть его другом.

Спасибо Джессу Д'Амико, Шону Маккулу, Йену Лэндсману и Тейлору за то, что увидели во мне ценного участника конференций и предоставили мне платформу для обучения. Спасибо Дейлу Риису за то, что помог многим людям легко освоить Laravel на заре появления этого фреймворка.

Спасибо всем, кто посвятил свое время и усилия написанию постов в блоге по фреймворку Laravel, особенно тем, кто был в числе первых: Эрику Барнсу, Крису Фидао, Мэтту Мачуге, Джейсону Льюису, Райану Табладе, Дрису Винтсу, Максусу Сургаю и многим другим.

Спасибо всем друзьям в Twitter, IRC и Slack, которые общались со мной на протяжении многих лет. Хотелось бы перечислить здесь всех, но я не буду, поскольку боюсь, что упущу из виду чье-то имя и потом буду сильно переживать. Вы все великолепны, и для меня большая честь регулярно общаться с вами.

Спасибо работавшему со мной редактору издательства O'Reilly Элли Макдональд и всем моим техническим редакторам: Киту Дамиани, Майклу Дайринде, Адаму Фэйрхольму и Майлсу Хайсону.

И конечно же, спасибо остальным членам моей семьи и друзьям, которые поддерживали меня прямо или косвенно, — родителям, братьям и сестрам, сообществу Гейнсвилла, другим владельцам компаний и авторам книг, участникам конференций и неподражаемому DCB. На этом, пожалуй, стоит остановиться, пока я не дошел до поименной признательности тем людям, которые готовили мне кофе в Starbucks.

Второе издание

Второе издание во многом схоже с первым, поэтому в силе остаются все предыдущие благодарности. В то же время мне помогли несколько новых людей. Моими техническими корректорами были Тейт Пеньяранда, Энди Свик, Мохамед Саид и Саманта Гейтц, а моим новым редактором от O'Reilly была Алисия Янг, которая помогла не оставлять работу над книгой, несмотря на произошедшие за последний год изменения в моей жизни и жизни сообщества Laravel. Мэтт Хакер из команды Atlas отвечал на все мои глупые вопросы о форматировании документации, созданной с помощью AsciiDoc, в том числе об удивительно сложном форматировании метода __().

И я не смог бы осилить второе издание без моего помощника Уилбура Пауэри. Уилбур без устали анализировал журналы изменений, запросы на включение и сообщения об изменениях за последние несколько лет, сверяя каждую функциональную возможность с содержанием книги, и даже тестировал в Laravel 5.7 (а затем и в 5.8) буквально каждый представленный в книге пример кода, чтобы я мог сосредоточиться на написании новых и обновлении старых разделов.

Кроме того, моя дочь Миа сейчас уже не в мамином животике, и ее радость, энергия, любовь и дух приключений теперь занимают не последнее место среди моих источников вдохновения.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Зачем использовать Laravel

На заре появления динамических веб-страниц процесс написания веб-приложения выглядел совершенно не так, как сейчас. Разработчик должен был не только реализовать уникальную бизнес-логику приложения, но и написать код всех других компонентов, которые так часто присутствуют на сайте, — модулей аутентификации пользователей, валидации ввода, доступа к базе данных, обработки шаблонов и т. д.

Сегодня у программистов есть десятки фреймворков для разработки приложений и тысячи легкодоступных компонентов и библиотек. Разработчики шутят, что к концу изучения одного фреймворка появляется три новых, которые предположительно лучше и должны его заменить.

В альпинизме наличие горы может быть достаточным основанием для ее покорения, однако, когда мы решаем, какой из фреймворков лучше использовать и стоит ли вообще это делать, нужно исходить из более веских оснований. Необходимо спросить у себя, зачем применять фреймворк, и в частности Laravel.

Для чего нужен фреймворк

Легко понять, почему PHP-разработчикам выгодно использовать отдельные компоненты или пакеты. В таком случае кто-то другой отвечает за разработку и поддержку изолированного фрагмента кода, имеющего четко определенные границы работы, и теоретически этот человек лучше разбирается в конкретном компоненте, чем вы.

Такие фреймворки, как Laravel, Symfony, Lumen или Slim, предварительно упаковывают коллекцию сторонних компонентов вместе со специфическим «клеем» фреймворка — файлами конфигурации, сервис-провайдерами, предписываемой структурой каталогов и программами начальной загрузки. Таким образом, преимущество от использования фреймворка в целом заключается в том, что для вас

кто-то принимает решения не только об отдельных компонентах, но и о том, *как они должны сочетаться*.

«Все своими руками»

Допустим, вы решили создать веб-приложение без использования фреймворка. С чего начать? Вероятно, приложение должно осуществлять маршрутизацию HTTP-запросов, и потому вам нужно оценить доступные библиотеки HTTP-запросов и ответов и выбрать одну из них. Вам также следует выбрать маршрутизатор. И конечно, вам необходимо настроить некий файл конфигурации маршрутов. Какой синтаксис следует использовать? Где его разместить? А что насчет контроллеров? Где их разместить и как должна производиться загрузка? Вероятно, надо использовать контейнер внедрения зависимостей для разрешения доступа к контроллерам и их зависимостям. Но какой?

Более того, ответив на все эти вопросы и успешно создав приложение, подумайте, как ваши решения повлияют на следующего разработчика. Представьте, что вам нужно поддерживать четыре или десять таких приложений на базе собственного фреймворка, помнить расположение контроллеров и синтаксис маршрутизации в каждом из них!

Согласованность и гибкость

Фреймворки избавляют от перечисленных выше проблем. Они дают тщательно продуманный ответ на вопрос о том, какой компонент следует использовать для той или иной цели, и гарантируют, что отдельные выбранные компоненты хорошо работают вместе. Благодаря тому, что фреймворки подразумевают выполнение ряда соглашений, при переходе к новому проекту разработчику требуется разбираться в меньшем количестве кода. Так, если вы знаете, как осуществляется маршрутизация в одном проекте Laravel, то знаете, как она работает во всех.

Когда кто-то советует вам создавать фреймворк для каждого проекта, подразумевается, что вы должны *контролировать*, что следует включать в «фундамент» вашего приложения, а что — нет. Хороший фреймворк не только обеспечит вам надежный «фундамент», но и позволит варьировать его состав. Как вы увидите далее, популярность Laravel во многом обусловлена именно такой возможностью.

Краткий экскурс в историю веб- и PHP-фреймворков

Чтобы ответить на вопрос, зачем использовать Laravel, нужно вспомнить, как появился данный фреймворк и что происходило до этого. Популярности Laravel предшествовало появление множества разнообразных фреймворков и ряд иных тенденций в сфере разработки на PHP и в других областях веб-разработки.

Ruby on Rails

В 2004 году Дэвид Хайнмайер Хансон выпустил первую версию Ruby on Rails, и практически каждый фреймворк для веб-приложений, выпущенный после, имел какие-то черты Rails.

Фреймворк Rails популяризировал MVC, API на базе RESTful и JSON, программирование по соглашениям, шаблон Active-Record и многие другие инструменты и соглашения, которые значительно повлияли на подход веб-программистов к созданию приложений, в частности увеличили скорость разработки.

Бум PHP-фреймворков

Большинство разработчиков понимало, что будущее за такими веб-фреймворками, как Rails, и на свет как грибы после дождя стали появляться новые PHP-фреймворки, в том числе откровенные копии Rails.

Первой ласточкой стал вышедший в 2005 году CakePHP, за которым последовали Symfony, CodeIgniter, Zend Framework и Kohana (ответвление от проекта CodeIgniter). В 2008 году был выпущен фреймворк Yii, а в 2010 году — Aura и Slim. В 2011 году появились FuelPHP и Laravel, которые не были простыми ответвлениями CodeIgniter, а предлагались в качестве альтернативы.

Многие из них сильно напоминали Rails, будучи ориентированными на применение объектно-реляционных отображений (ORM), MVC-структур и иных инструментов быстрой разработки. Другие, как, например, Symfony и Zend, были больше направлены на использование корпоративных шаблонов проектирования и электронной коммерции.

Преимущества и недостатки CodeIgniter

Фреймворки CakePHP и CodeIgniter были первыми в ряду PHP-фреймворков, созданных под влиянием Rails. CodeIgniter быстро завоевал известность и к 2010 году стал, пожалуй, самым популярным независимым PHP-фреймворком.

Фреймворк CodeIgniter был прост, удобен в применении, имел отличную документацию и активное сообщество. Однако он медленно развивался в плане использования современных технологий и шаблонов и по мере прогресса фреймворков и совершенствования инструментов PHP-разработки начал отставать и с точки зрения технологий, и с точки зрения предлагаемых по умолчанию функциональных возможностей. В отличие от многих других фреймворков CodeIgniter разрабатывался компанией, поэтому медленно внедрялась поддержка новых возможностей PHP 5.3, таких как пространства имен или использование GitHub, а позднее — Composer. К 2010 году создатель Laravel Тейлор Отвел настолько разочаровался в CodeIgniter, что решил написать собственный фреймворк.

Laravel 1, 2 и 3

Релиз бета-версии Laravel 1, написанной с нуля, состоялся в июне 2011 года. В ней присутствовали собственная ORM-система (Eloquent), маршрутизация на основе замыканий (навеянная фреймворком Ruby Sinatra), возможность расширения за счет использования модульной системы и хелперы (helper, вспомогательные функции) для форм, валидации, аутентификации и т. д.

Поначалу разработка Laravel велась очень быстро, Laravel 2 и 3 появились соответственно в ноябре 2011-го и феврале 2012 года. В них добавили контроллеры, модульное тестирование, инструмент командной строки, контейнер инверсии управления (Inversion of Control, IoC), отношения Eloquent и миграции.

Laravel 4

При создании Laravel 4 Тейлор полностью переписал весь фреймворк. К этому моменту Composer, менеджер пакетов для PHP, уже распространился настолько, что стал практически промышленным стандартом, и Отвел решил, что будет полезно переписать фреймворк в виде набора компонентов, которые можно распространять и упаковывать с помощью этого пакетного менеджера.

Тейлор разработал его под кодовым названием *Illuminate* и в мае 2013 года выпустил Laravel 4 с совершенно новой структурой. Теперь вам уже не предлагался для загрузки пакет с большей частью кода фреймворка; вместо этого Laravel подгружал большинство своих компонентов из фреймворка Symfony (компоненты которого можно свободно использовать в других фреймворках) и набора компонентов Illuminate с помощью менеджера Composer.

В Laravel 4 были также введены очереди, почтовый компонент, фасады и механизм заполнения баз данных первичными данными. Поскольку в Laravel теперь использовались компоненты фреймворка Symfony, объявили, что релизы Laravel будут выходить в соответствии (с небольшим отставанием) с шестимесячным графиком выпуска фреймворка Symfony.

Laravel 5

В ноябре 2014 года планировалось выпустить Laravel 4.3, но по мере продвижения разработки стало ясно, что изменения стоит выделить в более крупный релиз, которым стала версия Laravel 5 в феврале 2015 года.

В Laravel 5 была обновлена структура каталогов, удалены хелперы для форм и HTML, введены контрактные интерфейсы, множество новых представлений, библиотека Socialite для аутентификации в социальных сетях, библиотека Elixir для компиляции ресурсов, библиотека Scheduler для упрощения планирования задач cron, библиотека dotenv для более легкого управления средой, запросы форм

и совершенно новая реализация REPL-интерфейса (read — evaluate — print loop, цикл «чтение — вычисление — вывод»). С тех пор фреймворк рос в плане функциональности и зрелости, но никаких серьезных изменений, как в предыдущих версиях, уже не вносилось.

Чем уникален Laravel

Так что же делает Laravel уникальным? Почему для работы лучше выбрать его, а не какой-либо другой PHP-фреймворк? Ведь в каждом из них используются компоненты Symfony? Немного поговорим о том, что делает Laravel «именно тем, что нужно».

Философия Laravel

Чтобы понять, что выгодно отличает данный фреймворк от других, достаточно ознакомиться с его рекламными материалами и прочитать файлы `README`. Обратите внимание, какие слова употребляет Тейлор: сначала связанные со светом — «освещать», «искра», затем — «искусные мастера», «элегантные», а также «дыхание свежего воздуха», «новый старт» и, наконец, — «быстрый», «сверхсветовая скорость».

Двумя принципами данного фреймворка являются увеличение скорости разработки и повышение удобства разработчиков. Тейлор специально назвал интерфейс командной строки Artisan («искусный мастер»), чтобы подчеркнуть контраст с более утилитарными проектами. Еще в 2011 году в одном из своих ответов в сети StackExchange (<http://bit.ly/2dT5kmS>) Отвел признался, что порой тратит невероятно много времени — целые часы — на то, чтобы придать коду «красивый» вид — лишь для того, чтобы ему было приятно на него смотреть. Кроме того, он часто говорил, как важно предоставить разработчикам возможность проще и быстрее воплощать идеи, исключив ненужные препятствия для создания великолепных продуктов.

По сути, Laravel призван снабдить разработчиков инструментами и возможностями. Его задача — предоставить понятный, простой и красивый код и функции, позволяющие программистам быстро изучать, запускать, разрабатывать и писать код, отличающийся простотой, ясностью и долговечностью.

Принцип ориентации на разработчиков ясно прослеживается во всех материалах по Laravel. «Счастливые разработчики пишут лучший код», — написано в документации. Одно время неофициальный рекламный слоган фреймворка звучал как «счастье разработчика от загрузки до развертывания». Конечно, создатели любого инструмента или фреймворка могут сказать, что они заботятся о счастье программистов. Однако только в Laravel этому отводится *центральное* место, что оказало огромное влияние на общий стиль и процесс принятия решений. Если в других фреймворках на первое место может ставиться архитектурная чистота или совме-

стимость с целями и ценностями корпоративных групп разработчиков, то в Laravel основное внимание уделяется требованиям и запросам отдельного программиста. Это совсем не значит, что вы не сможете писать архитектурно чистые или корпоративные приложения с помощью Laravel; это лишь значит, что вам не придется жертвовать читабельностью и понятностью кодовой базы.

Как Laravel делает разработчиков счастливее

Легко сказать, что вы хотите сделать разработчиков счастливыми, и гораздо труднее претворить это в жизнь. Для этого нужно ответить на вопрос, что во фреймворке может осчастливить программистов или огорчить. В Laravel применяется несколько подходов, призванных облегчить жизнь разработчиков.

Во-первых, Laravel — это фреймворк для быстрой разработки. Он предельно прост в освоении и сводит к минимуму количество шагов между запуском нового приложения и его публикацией. Его компоненты упрощают разработку веб-приложения: от взаимодействия с базой данных и аутентификации до работы с очередями, электронной почтой и кэшем. Компоненты Laravel не только великолепно справляются со своей задачей, но и предоставляют единообразные API и предсказуемые структуры в рамках всего фреймворка: когда вы пробуете что-то новое в Laravel, вам нужно просто взять и запустить новую функцию.

Причем этот подход не ограничивается рамками фреймворка. Laravel предоставляет целую экосистему инструментов для создания и запуска приложений. У вас есть Homestead и Valet для локальной разработки, Forge для управления серверами и Envoyer для расширенного развертывания. Имеется набор дополнительных пакетов: Cashier для платежей и подписок, Echo для веб-сокетов, Scout для поиска, Passport для API-аутентификации, Dusk для тестирования клиентской части приложения (фронтенда), Socialite для аутентификации в социальных сетях, Horizon для отслеживания состояния очередей, Nova для создания панелей администратора и Spark для начальной загрузки SaaS-сервиса. Laravel призван избавить разработчиков от однообразных операций, чтобы они могли сосредоточиться на более творческих задачах.

В Laravel пропагандируется принцип «программирования по соглашениям»: если вы согласитесь применять предлагаемые значения по умолчанию, то выполните гораздо меньше работы, чем при использовании других фреймворков, которые требуют объявления всех настроек даже с рекомендованной конфигурацией. Создание проекта в Laravel занимает меньше времени, чем в большинстве других PHP-фреймворков.

Большое внимание уделяется простоте. Вы можете внедрять и имитировать зависимости, шаблоны Data Mapper и репозитории, Command Query Responsibility Segregation (разделение ответственности команд и запросов, CQRS) и любые другие более сложные архитектурные шаблоны. Но, если другие фреймворки, как правило, предлагают задействовать эти инструменты и структуры при создании каждого

проекта, Laravel, его документация и сообщество обычно изначально предлагают простейший вариант реализации на основе таких средств, как глобальная функция, фасад и ActiveRecord. Это позволяет создавать для решения задач предельно простое приложение, не отказываясь от возможности его применения в сложной среде.

Интересное отличие Laravel от других PHP-фреймворков в том, что его создатель и сообщество больше тяготеют к идеям Ruby and Rails и функциональным языкам программирования, чем к языку Java. В мире PHP очень сильна тенденция к многословности и сложности, характерная для Java-подобных компонентов PHP. Laravel же продвигает выразительные, динамичные и простые методы кодирования и возможности языка.

Сообщество Laravel

Если вам еще не приходилось сталкиваться с сообществом Laravel, то приготовьтесь открыть для себя нечто необычное. Одна из отличительных черт фреймворка Laravel, в немалой степени способствовавшая его росту и успеху, — наличие доброжелательного и готового делиться знаниями сообщества программистов. Начиная с видеоуроков Laracasts (<https://laracasts.com/>) от Джефффри Уэя, Laravel News (<https://laravel-news.com/>) и Slack-, IRC- и Discord-каналов, заканчивая постами в «Твиттере», блогами, подкастами и конференциями Laracon — все говорит о том, что у Laravel есть обширное и активное сообщество, в котором присутствуют и те, кто был в числе энтузиастов с самого первого дня, и те, кто только знакомится с этим фреймворком. И это не случайность.

«С самых первых дней фреймворка Laravel у меня была идея о том, что любой человек хочет чувствовать себя частью чего-то значимого. Это желание принадлежать к некоторой группе единомышленников заложено в нас самой человеческой природой. Поэтому, привнеся в веб-фреймворк немного личного подхода и достаточно активно взаимодействуя с сообществом, каждый сможет почувствовать себя частью целого».

Тейлор Отвел, интервью по продукту и поддержке

Уже в самом начале работы над Laravel Тейлор понял, что для успеха проект с открытым исходным кодом должен обладать двумя вещами: хорошей документацией и доброжелательным сообществом. И сегодня их можно смело назвать отличительными чертами Laravel.

Как работает Laravel

До сих пор мы говорили с вами исключительно об абстрактных вещах. А как насчет кода, спросите вы? Рассмотрим простое приложение (пример 1.1), чтобы вы могли увидеть, что такое работа с Laravel.

Пример 1.1. Программа Hello, World! в файле routes/web.php

```
<?php

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

Самое простое, что можно сделать в приложении Laravel, — определить маршрут и возвращать результат каждый раз, когда кто-то по нему переходит. Если вы инициализируете на компьютере новое приложение Laravel, определите маршрут из примера 1.1 и настройте работу сайта из *публичного* каталога, то получите полностью функциональную программу Hello, World! (рис. 1.1).

A screenshot of a web browser window. The address bar is empty. The main content area of the browser displays the text "Hello, World!" in a simple, black, sans-serif font. The browser's interface, including the address bar and some UI elements, is visible around the text.

Рис. 1.1. Возвращение строки Hello, World! с помощью Laravel

Как показывает пример 1.2, реализация этой программы с помощью контроллеров выглядит очень похоже.

Пример 1.2. Реализация программы Hello, World! с помощью контроллеров

```
// File: routes/web.php
<?php

Route::get('/', 'WelcomeController@index');

// File: app/Http/Controllers/WelcomeController.php
<?php

namespace App\Http\Controllers;

class WelcomeController extends Controller
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

Данная программа будет выглядеть во многом так же и в том случае, если вы разместите несколько приветствий в базе данных (пример 1.3).

Пример 1.3. Программа Hello, World! с выводом нескольких приветствий из базы данных

```
// File: routes/web.php
<?php

use App\Greeting;

Route::get('create-greeting', function () {
    $greeting = new Greeting;
    $greeting->body = 'Hello, World!';
    $greeting->save();
});

Route::get('first-greeting', function () {
    return Greeting::first()->body;
});

// File: app/Greeting.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Greeting extends Model
{
    //
}

// File: database/migrations/2015_07_19_010000_create_greetings_table.php
<?php

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

class CreateGreetingsTable extends Migration
{
    public function up()
    {
        Schema::create('greetings', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('body');
            $table->timestamps();
        });
    }

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

Если пример 1.3 кажется вам немного сложным, пока пропустите его. Мы подробно разберемся с этим кодом в следующих главах. Как можно понять уже сейчас, чтобы настроить миграции базы данных и модели, а затем извлечь записи, требуется написать всего несколько строк кода — все предельно просто!

Почему стоит выбрать Laravel

Почему стоит выбрать Laravel?

Потому что Laravel позволит вам воплощать свои идеи в жизнь без напрасно написанного кода, в соответствии с современными стандартами кодирования, при поддержке активного сообщества и при наличии мощной экосистемы инструментов.

И потому, что вы, дорогой разработчик, заслуживаете того, чтобы быть счастливым.

2

Настройка среды разработки для использования Laravel

Успех языка PHP во многом обусловлен тем, что сегодня трудно найти веб-сервер, который *не мог бы* обрабатывать PHP-код. В то же время сейчас к PHP-инструментам предъявляются более строгие требования, чем раньше. Чтобы обеспечить оптимальные условия для разработки с использованием Laravel, необходима согласованная локальная и удаленная серверная среда для кода. К счастью, в экосистеме Laravel для этого предусмотрено несколько инструментов.

Системные требования

Примеры в главе можно выполнить и на Windows, но потребуется прочитать десятки страниц специальных указаний и пояснений. Я оставляю эту задачу пользователям и сконцентрируюсь на разработчиках, использующих Unix/Linux/macOS.

Независимо от того, будете вы обслуживать свой сайт с помощью PHP и других инструментов на локальном компьютере, установив среду разработки на виртуальной машине с помощью Vagrant или Docker или используя пакет MAMP/WAMP/ХАМРР, для создания сайтов с помощью Laravel ваша среда разработки должна включать в себя следующее:

- ☐ PHP $\geq 7.1.3$ для Laravel версий 5.6–5.8, PHP $\geq 7.0.0$ для версии 5.5, PHP $\geq 5.6.4$ для версии 5.4, PHP 5.6.4–7.1.* для версии 5.3, PHP $\geq 5.5.9$ для версий 5.2 и 5.1;
- ☐ расширение PHP OpenSSL;
- ☐ расширение PHP PDO;
- ☐ расширение PHP Mbstring;
- ☐ расширение PHP Tokenizer;
- ☐ расширение PHP XML (Laravel 5.3 и выше);
- ☐ расширение PHP Ctype (Laravel 5.6 и выше);
- ☐ расширение PHP JSON (Laravel 5.6 и выше);
- ☐ расширение PHP BCMath (Laravel 5.7 и выше).

Composer

Какой бы вариант вы ни использовали, у вас должен быть установлен инструмент Composer (<https://getcomposer.org/>) — один из основных инструментов современной разработки на PHP. Composer — менеджер зависимостей для PHP, во многом такой же, как NPM для Node или RubyGems для Ruby. Вместе с тем, как и NPM, Composer служит «фундаментом» для выполнения большинства тестов, локальной загрузки сценариев, запуска сценариев установки и многого другого. Composer нужен для установки и обновления Laravel, а также загрузки внешних зависимостей.

Локальные среды разработки

Для многих проектов достаточно локальной среды разработки с самым простым набором инструментов. Если в вашей системе уже установлен пакет MAMP, WAMP или XAMPP, вам вряд ли потребуется устанавливать что-либо еще для запуска Laravel. Вы также можете запустить Laravel на встроенном веб-сервере PHP при условии, что в вашей системе установлена подходящая версия PHP.

Для начала этого достаточно, чтобы запускать PHP-код. Все, что свыше этого, я оставляю на ваше усмотрение.

Следует отметить, что Laravel предлагает две среды локальной разработки: Valet и Homestead. Если вы не знаете, какую выбрать, я рекомендую использовать Valet и в общих чертах ознакомиться с Homestead; однако обе эти среды по-своему полезны и заслуживают внимания.

Laravel Valet

Если вы решили использовать встроенный веб-сервер PHP, то самый простой вариант — обслуживать каждый сайт, указывая URL-адрес `localhost`. Если вы запустите команду `php -S localhost:8000 -t public` из корневой папки вашего сайта на базе Laravel, то встроенный веб-сервер PHP будет обслуживать сайт по адресу `http://localhost:8000/`. Вы также можете развернуть соответствующий сервер после настройки приложения, выполнив команду `php artisan serve`.

Если вы хотите привязать каждый из сайтов к определенному домену разработки, вам потребуется разобраться с файлом `hosts` операционной системы и использовать такой инструмент, как `dnsmasq` (<https://ru.wikipedia.org/wiki/Dnsmasq>). Попробуем что-нибудь попроще.

Если вы работаете в Mac (также имеются неофициальные выпуски для Windows и Linux), то Laravel Valet избавит вас от необходимости привязывать домены к папкам приложения. Valet установит `dnsmasq` и ряд сценариев PHP, после чего останется лишь набрать команду `laravel new myapp && open myapp.test`. Конечно, вам потребуется установить несколько инструментов с помощью менеджера пакетов

Homebrew и разобраться с соответствующей документацией, но в целом путь от начальной установки до обслуживания приложений — несколько простых шагов.

Установите среду разработки Valet — для этого прочтите в документации по адресу <http://bit.ly/2U7uy7b> актуальные инструкции по установке — и укажите один или несколько каталогов, в которых будут находиться ваши сайты. Так, я запустил команду `valet park` из каталога `~/Sites` на моем устройстве, где расположены все приложения, над которыми я работаю. Теперь вы можете открыть папку в своем браузере, просто добавив окончание `.test` к названию каталога.

Valet позволяет легко настроить обслуживание всех вложенных папок определенного каталога в формате `{folderName}.test` с помощью команды `valet park`; только одного каталога — командой `valet link`. Открыть для каталога домен, обслуживаемый этой средой, можно, введя команду `valet open`; настроить обслуживание сайта с использованием протокола HTTPS — `valet secure`; открыть туннель ngrok для совместного использования сайта — `valet share`.

Laravel Homestead

Homestead используется для настройки локальной среды разработки. Это инструмент конфигурирования, устанавливаемый поверх Vagrant — программного обеспечения для управления виртуальными машинами — и предоставляющий предварительно сконфигурированный образ виртуальной машины, который идеально настроен для разработки с помощью Laravel и имитирует наиболее типичный вариант эксплуатационной среды для сайтов Laravel. Homestead, вероятно, лучший вариант локальной среды разработки для программистов, работающих в Windows.

Документация по Homestead регулярно обновляется (<http://bit.ly/2FwQ7EZ>), поэтому ознакомьтесь с ней, если хотите узнать, как настроить и использовать этот инструмент.



Vessel

Это не официальный проект Laravel, но Крис Фидео из Servers for Hackers (<https://serversforhackers.com/>) и Shipping Docker (<https://shippingdocker.com/>) сделали простой инструмент создания сред Docker для разработки Laravel под названием Vessel (<https://vessel.shippingdocker.com/>). Посмотрите документацию, чтобы узнать больше.

Создание нового проекта Laravel

Существует два способа создания нового проекта, но они запускаются из командной строки. Первый способ: глобально установить установщик Laravel (с помощью менеджера пакетов Composer), а второй — использовать функцию `create-project` менеджера пакетов Composer.

Вы можете узнать об этих вариантах более подробно на странице документации по установке (<http://bit.ly/2HFzBFY>), но я бы порекомендовал использовать установщик Laravel.

Установка Laravel с помощью установщика Laravel

Если у вас глобально установлен менеджер пакетов Composer, то для Laravel достаточно выполнить следующую команду:

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

Далее можно легко развернуть новый проект, выполнив из командной строки такую команду:

```
laravel new projectName
```

Эта команда создаст в текущем каталоге подкаталог с именем `{projectName}` и установит в него пустой проект Laravel.

Установка Laravel с помощью функции create-project менеджера пакетов Composer

Можно воспользоваться функцией `create-project` менеджера пакетов Composer, которая позволяет создавать проекты с определенной структурой. Для этого выполните следующую команду:

```
composer create-project laravel/laravel projectName
```

В текущем каталоге также будет создан подкаталог с именем `{projectName}` с предварительным каркасом приложения.

Lambo: улучшенный вариант команды `laravel new`

Я написал простой сценарий Lambo (<http://bit.ly/2TCcQo8>) для автоматизации повторяющихся действий при создании нового проекта Laravel.

Lambo запускает команду `laravel new`, после чего регистрирует ваш код в репозитории Git, задает в качестве параметров доступа указанные в файле `.env` значения по умолчанию, открывает проект в браузере, (опционально) открывает его в редакторе и выполняет еще несколько полезных действий, касающихся создания проекта.

Вы можете установить Lambo с помощью команды `global require` менеджера пакетов Composer:

```
composer global require tightenco/lambo
```

После этого его можно будет использовать так же, как команду `laravel new`:

```
cd Sites  
lambo my-new-project
```

Структура каталогов Laravel

При открытии каталога с заготовкой приложения Laravel вы увидите следующие файлы и каталоги:

```
app/  
bootstrap/  
config/  
public/  
resources/  
routes/  
storage/  
tests/  
vendor/  
.editorconfig  
.env  
.env.example  
.gitattributes  
.gitignore  
artisan  
composer.json  
composer.lock  
package.json  
phpunit.xml  
readme.md  
server.php  
webpack.mix.js
```



Различные инструменты сборки в Laravel до версии 5.4

В проектах, созданных до Laravel 5.4, вы можете увидеть файл `gulpfile.js` вместо `webpack.mix.js`. Это означает, что проект использует Laravel Elixir (<http://bit.ly/2JCToYp>) вместо Laravel Mix (<http://bit.ly/2U4X09P>).

Кратко ознакомимся с ними.

Каталоги

Корневой каталог по умолчанию содержит следующие папки.

- ❑ **app** — здесь размещается основная часть вашего приложения — модели, контроллеры, команды и PHP-код домена.
- ❑ **bootstrap** — содержит файлы, которые Laravel использует для загрузки при каждом запуске.
- ❑ **config** — здесь находятся все конфигурационные файлы.
- ❑ **database** — содержит миграции баз данных, сидеры и фабрики.

- ❑ **public** — каталог, на который указывает сервер при обслуживании сайта. Содержит файл `index.php` — фронтальный контроллер, который запускает процесс начальной загрузки и маршрутизирует все запросы. Здесь также размещаются все публичные файлы: изображения, таблицы стилей, сценарии или загружаемые файлы.
- ❑ **resources** — здесь находятся файлы для других сценариев: представления, языковые файлы, а также (опционально) файлы исходного кода CSS/Sass/Less и файлы исходного кода JavaScript.
- ❑ **routes** — содержит все определения маршрутов как для HTTP-маршрутов, так и для «консольных маршрутов» или команд Artisan.
- ❑ **storage** — здесь находятся кэши, логи и скомпилированные системные файлы.
- ❑ **tests** — хранит модульные и интеграционные тесты.
- ❑ **vendor** — сюда устанавливаются зависимости менеджера пакетов Composer. Этот каталог игнорируется системой управления версиями Git (помечается как не контролируемый ею) в силу того, что действия Composer являются составной частью процесса развертывания на любых удаленных серверах.

Отдельные файлы

Корневой каталог также содержит следующие файлы.

- ❑ **.editorconfig** — инструкции для вашей среды разработки/текстового редактора в отношении предписываемых фреймворком стандартов кодирования (например, о размере отступов, кодировке и о том, следует ли обрезать конечные пробелы). Этот файл есть в любом приложении Laravel версии 5.5 или более новой.
- ❑ **.env** и **.env.example** — задают переменные среды (предположительно являются разными в разных средах и потому не регистрируются в системе управления версиями). **.env.example** — это шаблон, который дублируется каждой конкретной средой для создания собственного файла **.env**, игнорируемого системой управления версиями Git.
- ❑ **.gitignore** и **.gitattributes** — конфигурационные файлы системы управления версиями Git.
- ❑ **artisan** — позволяет запускать команды Artisan (см. главу 8) из командной строки.
- ❑ **composer.json** и **composer.lock** — конфигурационные файлы для Composer, при этом файл **composer.json** может редактироваться пользователем, а файл **composer.lock** — нет. Содержат некоторые базовые сведения о проекте, а также определяют его PHP-зависимости.
- ❑ **package.json** — файл, аналогичный **composer.json**, но предназначенный для ресурсов клиентской части и зависимостей системы сборки. Содержит указания

для менеджера пакетов NPM в отношении того, какие зависимости JavaScript следует подгрузить.

- ❑ `phpunit.xml` — конфигурационный файл для PHPUnit — инструмента, который Laravel использует для тестирования системы.
- ❑ `README.md` — файл Markdown, содержащий базовые сведения о фреймворке. Вы его не увидите, если используете установщик Laravel.
- ❑ `server.php` — резервный сервер, позволяющий выполнять предварительный просмотр приложения Laravel даже маломощным серверам.
- ❑ `webpack.mix.js` — конфигурационный (опциональный) файл для Mix. Если вы используете Elixir, то вместо этого файла увидите файл `gulpfile.js`. Эти файлы содержат указания для системы сборки в отношении способа компиляции и обработки ресурсов клиентской части.

Конфигурация

Основные настройки вашего приложения Laravel — настройки подключения к базе данных, параметры обработки очередей, электронной почты и т. д. — содержатся в файлах папки `config`. Каждый из этих файлов возвращает массив языка PHP, доступ к элементам которого осуществляется по конфигурационному ключу, состоящему из имени файла и всех ключей-потомков, разделенных точками (`.`).

Так, вы можете создать файл `config/services.php`, содержащий код следующего вида:

```
// config/services.php
<?php
return [
    'sparkpost' => [
        'secret' => 'abcdefg',
    ],
];
```

А затем получать доступ к этой переменной конфигурации с помощью выражения `config('services.sparkpost.secret')`.

Любые переменные конфигурации, которые должны быть разными у разных сред (и, следовательно, игнорироваться системой управления версиями), нужно перенести из этой папки в файлы `.env`. Допустим, вы хотите использовать для каждой среды разные ключи API Bugsnag. В таком случае настройте файл конфигурации так, чтобы он извлекал их из файла `.env`:

```
// config/services.php
<?php
return [
    'bugsnag' => [
        'api_key' => env('BUGSNAG_API_KEY'),
    ],
];
```

Хелпер `env()` извлекает значение из файла `.env`, содержащего нужный вам ключ. Соответственно, следует добавить его в файл `.env` (хранящий настройки данной среды) и `.env.example` (являющийся шаблоном для всех сред):

```
# В .env
BUGSNAG_API_KEY=oinfp9813410942
```

```
# В .env.example
BUGSNAG_API_KEY=
```

Файл `.env` уже будет содержать довольно много специфических для среды переменных с необходимой фреймворку информацией: сведениями об используемом драйвере электронной почты или настройках базы данных.



Использование функции `env()` вне файлов конфигурации

При вызове функции `env()` за пределами конфигурационных файлов могут быть недоступны некоторые возможности фреймворка `Laravel`, включая ряд функций кэширования и оптимизации.

Наилучший способ получения переменных среды — присвоение всех специфичных для среды значений элементам конфигурации. Считайте переменные среды в эти элементы конфигурации, а затем ссылайтесь на переменные конфигурации в любом месте приложения:

```
// config/services.php
return [
    'bugsnag' => [
        'key' => env('BUGSNAG_API_KEY'),
    ],
];

// В контроллере или где-либо еще
$bugsnag = new Bugsnag(config('services.bugsnag.key'));
```

Файл `.env`. Кратко рассмотрим содержимое файла `.env` по умолчанию. Список ключей может немного варьироваться в зависимости от используемой версии приложения, но в случае `Laravel 5.8` он выглядит так, как показано в примере 2.1.

Пример 2.1. Переменные среды по умолчанию в `Laravel 5.8`

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack

DB_CONNECTION=mysql
```

```
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1

MIX_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
MIX_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

Я не буду описывать назначение всех ключей, поскольку многие из них представляют собой группы аутентификационных данных для различных сервисов (Pusher, Redis, DB, Mail). В то же время стоит обратить внимание на две важные переменные среды, о которых вы должны знать.

- ❑ **APP_KEY** — случайно сгенерированная строка для шифрования данных. Если этот ключ будет пустым, вы можете столкнуться с ошибкой «Не указан ключ шифрования приложения». Тогда запустите команду `php artisan key:generate`, и Laravel сгенерирует ключ для вас.
- ❑ **APP_DEBUG** — логическое значение, определяющее, должны ли пользователи этого экземпляра вашего приложения видеть ошибки отладки, — хорошо подходит для локальных и промежуточных сред и абсолютно не подходит для эксплуатационной.

Остальным не связанным с аутентификацией параметрам (**BROADCAST_DRIVER**, **QUEUE_CONNECTION** и т. д.) присваиваются значения по умолчанию, обеспечивающие минимально возможную зависимость от внешних сервисов. Рекомендуется начинающим разработчикам.

Для большинства проектов после запуска приложения нужно изменить параметры конфигурации базы данных. Поскольку я использую Laravel Valet, то присваиваю параметру `DB_DATABASE` имя своего проекта, параметру `DB_USERNAME` — значение `root`, а параметру `DB_PASSWORD` — пустую строку:

```
DB_DATABASE=myProject
DB_USERNAME=root
DB_PASSWORD=
```

Затем я создаю базу данных с таким же, как у проекта, именем в том клиенте MySQL, который предпочитаю использовать. Готово.

Завершение настройки

На этом подготовку к работе пустого проекта Laravel можно считать завершенной. Остается лишь запустить команду `git init`, зарегистрировать пустые файлы с помощью команд `git add` и `git commit` — и можно приступать к кодированию! Если вы используете Valet, то можете сразу увидеть, как фактически выглядит ваш сайт в браузере, выполнив следующие команды:

```
laravel new myProject && cd myProject && valet open
```

Начиная новый проект, я выполняю следующие команды:

```
laravel new myProject
cd myProject
git init
git add .
git commit -m "Initial commit"
```

Поскольку я размещаю свои сайты в папке `~/Sites`, выбранной в качестве основного каталога среды Valet, после выполнения этих команд в браузере сразу же доступно имя `myProject.test`. Затем мне остается отредактировать файл `.env` так, чтобы он указывал на конкретную базу данных, добавить ее в свое приложение для работы с MySQL, и я готов кодировать! А если вы решите использовать Lambo, то все будет сделано автоматически.

Тестирование

В последующих главах в заключительном разделе «Тестирование» я буду показывать, как следует писать тесты для рассмотренных в главе функций. Поскольку в этой главе мы не рассматривали какие-либо тестируемые возможности, просто немного поговорим о тестировании (подробнее о написании и запуске тестов в Laravel вы прочитаете в главе 12).

По умолчанию фреймворк добавляет PHPUnit в качестве зависимости и настроен на запуск тестов, содержащихся в любом файле, который размещен в каталоге `tests` и имеет окончание `Test.php` (например, `tests/UserTest.php`).

Таким образом, самый легкий способ написания тестов состоит в том, чтобы создать в каталоге `tests` файл с именем, оканчивающимся на `Test.php`. И самый простой способ запуска — выполнить в командной строке команду `./vendor/bin/phpunit` (находясь в корневой папке проекта).

Если для каких-либо тестов требуется доступ к базе данных, то тесты следует запускать на том компьютере, где размещена ваша база данных, — поэтому, если вы размещаете свою базу данных в Vagrant, не забудьте подключиться к Vagrant-box по протоколу `ssh` и запустить свои тесты из него. Об этом и многом другом подробно написано в главе 12.

Следует отметить, что если вы читаете эту книгу впервые, то в разделах, посвященных тестированию, встретите незнакомый вам синтаксис и описание новых возможностей тестирования. Если вы не сможете разобраться в коде какого-либо из этих разделов, просто пропустите его и вернитесь уже после прочтения главы о тестировании.

Резюме

Поскольку Laravel является PHP-фреймворком, его очень просто обслуживать локально. Он также предоставляет два инструмента для управления вашей локальной разработкой: более простой — Valet, который использует для загрузки зависимостей локальную машину, и предварительно настроенную конфигурацию Vagrant под названием Homestead. Laravel применяет менеджер пакетов Composer и может устанавливаться с его помощью. По умолчанию загружается ряд папок и файлов, отражающих соглашения фреймворка и его взаимосвязи с другими инструментами с открытым исходным кодом.

3

Маршрутизация и контроллеры

Важная функция любого фреймворка для создания веб-приложений — прием запросов от пользователя и возвращение ответов, как правило, посредством протокола HTTP(S). Это означает, что при изучении нужно сначала разобраться с определением маршрутов приложения. Без маршрутов у вас не будет возможности взаимодействовать с конечным пользователем.

В этой главе мы рассмотрим работу с маршрутами в Laravel. Вы узнаете, как их определять и связывать с выполняемым кодом и как удовлетворять разнообразные потребности в маршрутизации с помощью соответствующих инструментов.

Краткое введение в MVC, команды HTTP и REST

Все, о чем мы будем говорить в этой главе, относится к способу организации MVC-приложения, а во многих примерах используются REST-подобные имена и команды. Поэтому кратко рассмотрим и то и другое.

Что такое MVC

Паттерн MVC включает в себя три основных понятия.

- ❑ *Модель (model)*. Это представление отдельно таблицы базы данных (или записи этой таблицы) — например, «Компания» или «Собака».
- ❑ *Представление/вид (view)*. Это шаблон для представления данных конечному пользователю: шаблон страницы авторизации с некоторым набором HTML-, CSS- и JavaScript-кода.
- ❑ *Контроллер (controller)*. Подобно регулирующему дорожное движение полицейскому, контроллер принимает HTTP-запросы от браузера, получает нужные данные от базы данных и других механизмов хранения, осуществляет валидацию пользовательского ввода и, наконец, возвращает пользователю ответ.

На рис. 3.1 можно увидеть, что конечный пользователь сначала взаимодействует с контроллером, отправляя HTTP-запрос с помощью браузера. Контроллер в ответ на этот запрос может записать данные и/или извлечь данные из модели (базы данных). После этого он отправляет данные в представление, которое возвращается конечному пользователю для отображения в браузере.

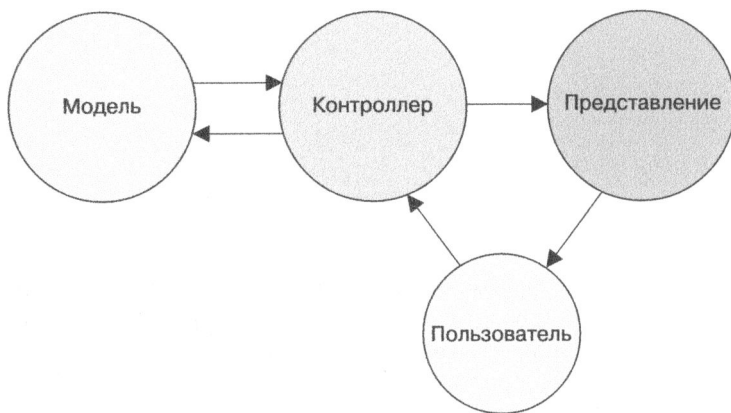


Рис. 3.1. Паттерн MVC

Поскольку некоторые варианты использования Laravel не «вписываются» в это упрощенное понимание архитектуры приложения, не стоит слишком заикливаться на паттерне MVC. Но, если будете знать его, вам будет легче понять данную главу.

HTTP-команды

Самые часто используемые HTTP-команды — GET и POST; следующие — PUT и DELETE. Еще есть HEAD, OPTIONS, PATCH и две практически не используемые в обычной веб-разработке: TRACE и CONNECT.

Команды имеют следующее назначение.

- ☐ GET — запрашивает ресурс (или список ресурсов).
- ☐ HEAD — запрашивает версию GET-ответа, содержащую только заголовок.
- ☐ POST — создает ресурс.
- ☐ PUT — перезаписывает ресурс.
- ☐ PATCH — модифицирует ресурс.
- ☐ DELETE — удаляет ресурс.
- ☐ OPTIONS — запрашивает у сервера список команд, разрешенных для конкретного URL-адреса.

В табл. 3.1 представлен список действий, доступных для контроллера ресурсов (подробнее о них написано в подразделе «Контроллеры ресурсов» на с. 71). Каждое действие подразумевает вызов нужного шаблона URL-адреса с помощью конкретной команды.

Таблица 3.1. Методы контроллеров ресурсов Laravel

Команда	URL	Метод контроллера	Имя	Описание
GET	tasks	index()	tasks.index	Показать все задачи
GET	tasks/create	create()	tasks.create	Показать форму создания задачи
POST	tasks	store()	tasks.store	Принять подачу формы из формы создания задачи
GET	tasks/{task}	show()	tasks.show	Показать одну задачу
GET	tasks/{task}/edit	edit()	tasks.edit	Отредактировать одну задачу
PUT/PATCH	tasks/{task}	update()	tasks.update	Принять подачу формы из формы редактирования задачи
DELETE	tasks/{task}	destroy()	tasks.destroy	Удалить одну задачу

Что такое REST

Подробнее рассмотрим REST в разделе «Базовые сведения о REST-подобных API на базе JSON» на с. 350, а пока отмечу, что это архитектурный стиль для создания API. В данной книге соответствие стилю REST понимается как наличие таких характеристик, как:

- ❑ ориентация на обработку одного основного ресурса за раз (например, `tasks`);
- ❑ организация взаимодействий с использованием URL-адресов с предсказуемой структурой и HTTP-команд (наподобие приведенных в табл. 3.1);
- ❑ возвращение, а часто и запрашивание данных в формате JSON.

Это далеко не все: в большинстве случаев под RESTful в книге будет подразумеваться использование шаблонной структуры URL-адресов, чтобы можно было делать предсказуемые вызовы, например, вида `GET /tasks/14/edit` для страницы редактирования. Это важно даже в том случае, если вы не собираетесь создавать API, поскольку, как можно видеть из табл. 3.1, структуры маршрутизации Laravel тоже организованы по REST-подобному паттерну.

В REST-подобных API используется точно такая же структура, лишь с тем отличием, что там нет маршрутов для создания и редактирования, поскольку API предоставляют только действия, но не страницы, выполняющие подготовку к действиям.

Определения маршрутов

В приложении Laravel вы будете определять свои веб-маршруты в файле `routes/web.php`, а API-маршруты — в файле `routes/api.php`. Веб-маршруты — это маршруты, по которым будут переходить ваши конечные пользователи, а API-маршруты — маршруты для вашего API, если вы используете таковой. Пока сосредоточимся на маршрутах в файле `routes/web.php`.



Расположение файла маршрутов в Laravel до версии 5.3

В проектах, использующих версии Laravel до 5.3, будет только один файл маршрутов — `app/Http/routes.php`.

Простейший способ определения маршрута — сопоставление пути (например, `/`) с замыканием, как показано в примере 3.1.

Пример 3.1. Простейший способ определения маршрута

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

ЧТО ТАКОЕ ЗАМЫКАНИЕ

Замыкания — это используемая в PHP разновидность анонимных функций. Их можно присваивать переменной, передавать как объект, в качестве параметра другим функциям и методам или даже сериализовывать.

Здесь мы определили, что, если кто-либо перейдет по адресу `/` (то есть в корневую папку вашего домена), маршрутизатор Laravel запустит определенное там замыкание и вернет результат. Заметьте, что мы именно возвращаем содержимое с помощью команды `return`, а не выводим его на экран командой `echo` или `print`.



Кратко о Middleware

Вероятно, вы хотите спросить, почему мы возвращаем строку `'Hello, World!'` с помощью команды `return`, вместо того чтобы вывести ее командой `echo`?

Есть много объяснений, но вот самый простой ответ — цикл запроса и ответа фреймворка Laravel заключен в большое количество оберток, в число которых входит и `middleware`. Поэтому не следует сразу же отправлять результат браузеру после выполнения замыкания маршрута или метода контроллера. Возврат содержимого командой `return` обеспечивает его прохождение через весь стек ответов и `middleware` до его возвращения пользователю.

Многие простые сайты могут быть полностью определены в файле веб-маршрутов. С помощью нескольких простых маршрутов GET в сочетании с некоторыми шаблонами вы можете легко обслуживать классический сайт (пример 3.2).

Пример 3.2. Пример сайта

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



Статические вызовы

Если у вас большой опыт разработки на PHP, вы удивитесь при виде статических вызовов в классе Route. Это не статический метод, а сервис-локатор, использующий фасады Laravel, которые мы рассмотрим в главе 11.

Если вы предпочитаете не использовать фасады, то перепишите эти определения следующим образом:

```
$router->get('/', function () {  
    return 'Hello, World!';  
});
```

Команды маршрутов

Вы могли заметить, что мы использовали выражение `Route::get()` в определениях маршрутов. Тем самым мы дали Laravel указание сопоставлять маршруты, только когда в HTTP-запросе определено действие GET. Но что, если это запрос POST формы или запросы PUT или DELETE некоторого JavaScript-кода? Как показано в примере 3.3, в определении маршрута можно вызывать и несколько других методов.

Пример 3.3. Команды маршрутов

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

```

    // Обслуживаем кого-то, отправляющего запрос POST на этот маршрут
});

Route::put('/', function () {
    // Обслуживаем кого-то, отправляющего запрос PUT на этот маршрут
});

Route::delete('/', function () {
    // Обслуживаем кого-то, отправляющего запрос DELETE на этот маршрут
});

Route::any('/', function () {
    // Обслуживаем запрос любой команды по этому маршруту
});

Route::match(['get', 'post'], '/', function () {
    // Обслуживаем запросы GET или POST по этому маршруту
});

```

Обработка маршрутов

Как вы, вероятно, догадались, передача замыкания в определение маршрута не единственный способ обеспечить его распознавание. Хотя замыкания — это быстро и просто, по мере увеличения вашего приложения будет все сложнее размещать логику маршрутизации в одном файле. Кроме того, приложения, использующие замыкания маршрутов, не могут задействовать возможности Laravel по кэшированию маршрутов (подробнее об этом — позже), позволяющие экономить сотни миллисекунд на обработке каждого запроса.

Еще один способ: вместо замыкания передавать имя контроллера и метод в виде строки, как показано в примере 3.4.

Пример 3.4. Маршруты вызывают методы контроллера

```
Route::get('/', 'WelcomeController@index');
```

Этот код дает фреймворку указание передавать направляемые запросы методу `index()` контроллера `App\Http\Controllers>WelcomeController`. Он получит такие же параметры и будет обработан так же, как было бы обработано замыкание.

СИНТАКСИС ССЫЛОК НА КОНТРОЛЛЕР/МЕТОД В LARAVEL

Согласно принятому в Laravel соглашению ссылаться на конкретный метод конкретного контроллера нужно следующим образом: *ControllerName@methodName*. Хотя этот формат часто играет роль лишь неформальной договоренности в отношении способа коммуникации, он также используется и в реальных привязках, как показано в примере 3.4. Laravel выделяет в строке сегменты до и после символа `@` и использует

их для идентификации контроллера и метода. В Laravel 5.7 также был введен «кортежный» синтаксис (`Route::get('/', [WelcomeController::class, 'index'])`), имя по-прежнему часто передается строкой вида *ControllerName@methodName*.

Параметры маршрутов

Если определяемый вами маршрут имеет параметры — сегменты в структуре URL-адреса, то их можно легко указать в маршруте и далее передавать замыканию (пример 3.5).

Пример 3.5. Параметры маршрута

```
Route::get('users/{id}/friends', function ($id) {  
    //  
});
```

Вы также можете сделать параметры маршрута необязательными, добавив знак вопроса (?) после имени параметра, как показано в примере 3.6. В этом случае вы должны указать значение по умолчанию для соответствующей переменной.

Пример 3.6. Необязательные параметры маршрута

```
Route::get('users/{id?}', function ($id = 'fallbackId') {  
    //  
});
```

Вы можете использовать регулярные выражения для определения того, что маршрут должен сопоставляться только в случае, если параметр удовлетворяет конкретным требованиям, как показано в примере 3.7.

Пример 3.7. Наложение на маршрут ограничений с помощью регулярных выражений

```
Route::get('users/{id}', function ($id) {  
    //  
})->where('id', '[0-9]+');  
  
Route::get('users/{username}', function ($username) {  
    //  
})->where('username', '[A-Za-z]+');  
  
Route::get('posts/{id}/{slug}', function ($id, $slug) {  
    //  
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

Как вы, наверное, догадались, сопоставление не выполняется при переходе по пути, соответствующему строке маршрута, но при этом регулярное выражение не соответствует параметру. Поскольку сопоставление выполняется сверху вниз, при сравнении пути `users/abc` в примере 3.7 будет пропущено первое замыкание, но получено соответствие со вторым, с которым маршрут и будет сопоставлен.

Но путь `posts/abc/123` не будет соответствовать ни одному из замыканий, поэтому мы получим ошибку 404 (не найдено).

ВЗАИМОСВЯЗЬ МЕЖДУ ИМЕНАМИ ПАРАМЕТРОВ МАРШРУТА И ИМЕНАМИ ПАРАМЕТРОВ МЕТОДА ЗАМЫКАНИЯ/КОНТРОЛЛЕРА

Как видно из примера 3.5, использование одинаковых имен для параметров маршрута (`{id}`) и параметров метода, внедряемого в определение маршрута (`function ($id)`), — распространенная практика. Но так ли это нужно?

Если вы не используете привязку модели маршрута, о которой речь пойдет дальше, то нет. Единственное, что определяет, с каким параметром маршрута должен сопоставляться тот или иной параметр метода, — это их порядок (слева направо), как можно видеть в следующем примере:

```
Route::get('users/{userId}/comments/{commentId}', function (
    $thisIsActuallyTheUserId,
    $thisIsReallyTheCommentId
) {
    //
});
```

Но тот факт, что вы *можете* использовать разные имена, еще не означает, что вы *должны* так поступать. Я рекомендую указывать одинаковые, чтобы не усложнять жизнь последующим разработчикам.

Имена маршрутов

Ссылаться на маршруты в другом месте приложения проще всего, указывая соответствующий путь. Глобальный хелпер `url()` упрощает создание таких ссылок в представлениях; как он используется, показано в примере 3.8. Хелпер `url()` дополняет маршрут полным доменным именем сайта.

Пример 3.8. Хелпер `url()`

```
<a href="<?php echo url('/'); ?>">
// Выводит <a href="http://myapp.com/">
```

В Laravel также можно присваивать каждому маршруту имя, что позволяет ссылаться на него без явного указания URL-адреса. Этот способ удобен тем, что вы можете давать простые псевдонимы сложным маршрутам и вам не требуется переписывать ссылки в клиентской части приложения при изменении путей (пример 3.9).

Пример 3.9. Определение имен маршрутов

```
// Определение маршрута с использованием метода name() в файле routes/web.php:
Route::get('members/{id}', 'MembersController@show')->name('members.show');
```

```
// Обращение к маршруту в представлении с помощью функции route():
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

Пример демонстрирует несколько новых концепций. Мы используем «текущее» определение маршрута для добавления имени, указав `name()` после `get()`. Метод `name()` позволяет нам присвоить маршруту короткий псевдоним, позволяющий легко ссылаться на него в другом месте приложения.



Определение пользовательских маршрутов в Laravel 5.1

В Laravel 5.1 нет возможности использования «текучих» определений маршрутов. Вместо этого потребуется передать массив второму параметру определения пути; подробное описание этого способа см. в документации по Laravel (<http://bit.ly/2UZm1Aw>). Пример 3.9 будет выглядеть так:

```
Route::get('members/{id}', [
    'as' => 'members.show',
    'uses' => 'MembersController@show',
]);
```

В нашем примере мы назвали маршрут `members.show`, что соответствует принятому в Laravel соглашению в отношении именования и представлений: *resourcePlural.action*.

СОГЛАШЕНИЯ В ОТНОШЕНИИ ИМЕНОВАНИЯ МАРШРУТОВ

Хотя маршрут можно назвать как угодно, общепринятая практика — указывать составное имя, включающее в себя название ресурса во множественном числе, точку и наименование действия. Так, для ресурса `photo` часто используются следующие пути:

```
photos.index
photos.create
photos.store
photos.show
photos.edit
photos.update
photos.destroy
```

Подробнее об этих соглашениях написано в подразделе «Контроллеры ресурсов» на с. 71.

Пример также демонстрирует применение хелпера `route()`. Как и `url()`, он упрощает обращение к именному маршруту в представлениях. Если у пути нет параметров, вы можете просто передать его имя (`route('members.index')`) и получить строку маршрута (<http://myapp.com/members>). Если у него есть параметры, передайте их как массив в качестве второго параметра, как в примере 3.9.

Я рекомендую ссылаться на маршруты с помощью имени, а не пути и использовать `route()`, а не `url()`. Иногда этот способ может быть несколько громоздким — в случае

если вы работаете с несколькими поддоменами, — но обеспечивает невероятный уровень гибкости, позволяя очень легко изменять структуру маршрутизации приложения.

ПЕРЕДАЧА ПАРАМЕТРОВ МАРШРУТОВ В ХЕЛПЕР `ROUTE()`

Если у вашего маршрута есть параметры (например, `users/id`), то их нужно определить при использовании хелпера `route()` для создания ссылки на маршрут.

Есть несколько разных способов передачи. Допустим, мы определили маршрут как `users/userId/comments/commentId`. Если идентификатор пользователя равен 1, а идентификатор комментария — 2, то нам доступны следующие варианты.

Вариант 1:

```
route('users.comments.show', [1, 2])  
// http://myapp.com/users/1/comments/2
```

Вариант 2:

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2])  
// http://myapp.com/users/1/comments/2
```

Вариант 3:

```
route('users.comments.show', ['commentId' => 2, 'userId' => 1])  
// http://myapp.com/users/1/comments/2
```

Вариант 4:

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2, 'opt' => 'a'])  
// http://myapp.com/users/1/comments/2?opt=a
```

Как видите, значения массива без ключей присваиваются в соответствии с порядком их следования. Значения массива с ключами сопоставляются с параметрами маршрута, соответствующими их ключам, с добавлением неиспользованных параметров в качестве запроса.

Группы маршрутов

Часто группа маршрутов имеет нечто общее — определенное требование аутентификации, префикс пути или, возможно, пространство имен контроллера. Многократное определение этих общих характеристик не только становится неэффективной тратой сил, но и может привести к загромождению файла, сделав малозаметными определенные структуры приложения.

Группы маршрутов позволяют собрать несколько путей вместе и однократно применить к ним общие параметры конфигурации, устранив ненужное дублирование. Кроме того, они являются визуальными подсказками для последующих разработчиков (а также вас самих) о том, что эти маршруты сгруппированы вместе.

Чтобы сгруппировать два или более пути, следует «окружить» их определения группой маршрутов, как показано в примере 3.10. При этом вы просто передаете замыкание определению группы и указываете такие маршруты внутри этого замыкания.

Пример 3.10. Определение группы маршрутов

```
Route::group(function () {  
    Route::get('hello', function () {  
        return 'Hello';  
    });  
    Route::get('world', function () {  
        return 'World';  
    });  
});
```

По умолчанию группа маршрутов ничего не делает. В примере 3.10 можно было бы с тем же успехом отделить часть маршрутов комментариями.

Middleware

Вероятно, наиболее распространенным использованием групп маршрутов является применение к ним middleware¹. Оно будет подробно рассмотрено в главе 10, но надо отметить, что оно применяется в Laravel для аутентификации пользователей и недопущения посещения гостевыми пользователями определенных частей сайта.

В примере 3.11 мы создаем группу маршрутов вокруг представлений `dashboard` и `account` и применяем к ним middleware `auth`. В данном случае это означает, что для доступа к панели мониторинга и странице учетной записи пользователи должны пройти аутентификацию.

Пример 3.11. Запрещение доступа к группе маршрутов для пользователей, не прошедших аутентификацию

```
Route::middleware('auth')->group(function() {  
    Route::get('dashboard', function () {  
        return view('dashboard');  
    });  
    Route::get('account', function () {  
        return view('account');  
    });  
});
```

¹ «Связующее программное обеспечение» (также переводится как «промежуточное программное обеспечение», «программное обеспечение среднего слоя», «подпрограммное обеспечение», «межплатформенное программное обеспечение») — широко используемый термин, означающий слой или комплекс технологического программного обеспечения для налаживания взаимодействия между различными приложениями, системами, компонентами («Википедия»). — *Примеч. ред.*



Изменение групп маршрутов до Laravel 5.4

Так же как «текущее» определение маршрута появилось только в Laravel 5.2, «текущий» способ применения таких модификаторов, как `middleware`, префиксы, домены и т. д., к группам маршрутов стал возможен только в версии 5.4.

Вот как будет выглядеть пример 3.11 в Laravel 5.3 и более ранних версиях:

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('dashboard', function () {
        return view('dashboard');
    });
    Route::get('account', function () {
        return view('account');
    });
});
```

Применение middleware в контроллерах

Понятнее и проще привязать `middleware` к маршрутам в контроллере, а не в определении маршрута. Для этого нужно вызвать метод `middleware()` в конструкторе контроллера. Методу `middleware()` передается строка с именем `middleware`; опционально к нему можно добавить метод-модификатор (`only()` или `except()`), определяющий, на что будет распространяться действие `middleware`:

```
class DashboardController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('admin-auth')
            ->only('editUsers');

        $this->middleware('team-member')
            ->except('editUsers');
    }
}
```

Обратите внимание: если вам часто приходится указывать модификаторы `only` и `except`, возможно, вам следует использовать еще один контроллер для маршрутов-исключений.

Ограничение частоты запросов

5.2 Если вам нужно ограничить доступ пользователей к определенному маршруту (маршрутам) некоторым предельным количеством запросов за единицу времени (что называется *ограничением частоты запросов* и больше характерно для API), то в версии 5.2 и выше для этой цели есть `middleware`. Добавьте `throttle`, который принимает два параметра: количество разрешенных пользователю попыток и время ожидания в минутах до сброса счетчика попыток. Применение показано в примере 3.12.

Пример 3.12. Применение middleware для ограничения частоты запросов к маршруту

```
Route::middleware('auth:api', 'throttle:60,1')->group(function () {
    Route::get('/profile', function () {
        //
    });
});
```

Динамическое ограничение частоты запросов. Если вам нужно, чтобы к разным пользователям применялись разные ограничения, можно дать указание `throttle` извлекать количество попыток (его первый параметр) из пользовательской модели Eloquent. Вместо того чтобы передавать в middleware сумму попыток в качестве первого параметра, отправьте имя атрибута модели Eloquent, и по нему будет определяться, какое ограничение числа запросов установлено для пользователя.

Так, если пользовательская модель имеет атрибут `plan_rate_limit`, это middleware можно использовать так: `throttle:plan_rate_limit,1`.

ЧТО ТАКОЕ ELOQUENT

Мы подробно поговорим об Eloquent, доступе к базам данных и генераторе запросов фреймворка в главе 5, но уже сейчас будет полезно иметь представление о том, что они собой представляют.

Eloquent — это используемый в Laravel ORM баз данных на основе шаблона ActiveRecord, позволяющий легко связать класс (модель) `Post` с таблицей `posts` базы данных и получить все записи с помощью вызова вида `Post::all()`.

Генератор запросов — это инструмент, позволяющий выполнять вызовы вида `Post::where('active', true)->get()` или `DB::table('users')->all()`. То есть он дает возможность выполнять *генерирование* запросов, составляя последовательные цепочки методов.

Префиксы путей

Если часть группы ваших маршрутов имеет общий сегмент пути, например, если панель мониторинга вашего сайта имеет префикс `/dashboard`, то вы можете упростить структуру, добавив префикс к группе маршрутов (пример 3.13).

Пример 3.13. Добавление префикса к группе маршрутов

```
Route::prefix('dashboard')->group(function () {
    Route::get('/', function () {
        // Обрабатывает путь /dashboard
    });
    Route::get('users', function () {
        // Обрабатывает путь /dashboard/users
    });
});
```

У каждой группы с префиксом также есть маршрут /, указывающий на корневой каталог — в примере 3.13 это папка /dashboard.

Запасные маршруты

В Laravel до версии 5.6 можно было задать «запасной маршрут» (размещаемый в конце файла маршрутов), чтобы перехватывать все несопоставленные пути:

```
Route::any('{anything}', 'CatchAllController')->where('anything', '*');
```

5.6 В Laravel 5.6+ вы можете вместо этого использовать метод `Route::fallback()`:

```
Route::fallback(function () {
    //
});
```

Поддоменная маршрутизация

Поддоменная маршрутизация определяется так же, как добавление префикса к группе маршрутов, с тем отличием, что вместо префикса здесь указывается имя поддомена. Такая маршрутизация имеет две основные области применения. Во-первых, ее можно использовать для представления разных разделов приложения (или даже разных приложений) в разных поддоменах. Как это можно сделать, показано в примере 3.14.

Пример 3.14. Поддоменная маршрутизация

```
Route::domain('api.myapp.com')->group(function () {
    Route::get('/', function () {
        //
    });
});
```

Во-вторых, иногда требуется передавать часть поддомена в качестве параметра, как показано в примере 3.15. Это часто делается в случае применения мультиарендности (например, в сервисах Slack или Harvest каждая компания получает собственный поддомен вида `tighsten.slack.co`).

Пример 3.15. Параметризованная маршрутизация поддоменов

```
Route::domain('{account}.myapp.com')->group(function () {
    Route::get('/', function ($account) {
        //
    });
    Route::get('users/{id}', function ($account, $id) {
        //
    });
});
```

Обратите внимание, что любые параметры для группы передаются в методы сгруппированных маршрутов в качестве первого параметра (параметров).

Префиксы пространства имен

Когда вы группируете маршруты по поддомену или префиксу, их контроллеры обычно имеют похожее пространство имен РНР. Так, в примере с панелью мониторинга все контроллеры маршрутов панели мониторинга могут находиться в пространстве имен `Dashboard`. Используя префикс пространства имен для группы маршрутов, как показано в примере 3.16, вы можете избежать длинных ссылок на контроллеры в таких группах, как `Dashboard/UsersController@index` и `Dashboard/PurchasesController@index`.

Пример 3.16. Префиксы пространства имен для групп маршрутов

```
// App\Http\Controllers\UsersController
Route::get('/', 'UsersController@index');

Route::namespace('Dashboard')->group(function () {
    // App\Http\Controllers\Dashboard\PurchasesController
    Route::get('dashboard/purchases', 'PurchasesController@index');
});
```

Префиксы имен

Возможности использования префиксов на этом не заканчиваются. Обычно имена маршрутов отражают цепочку наследования элементов пути, поэтому путь `users/comments/5` будет обслуживаться маршрутом с именем `users.comments.show`. В таком случае обычно используют группу для всех маршрутов, которые находятся под ресурсом `users.comments`.

Подобно тому как мы можем добавлять префиксы к сегментам URL-адреса и пространствам имен контроллеров, мы можем добавлять строковые префиксы и к названиям маршрутов. Используя префиксы имен для групп маршрутов, можно определить общий строковый префикс для всех наименований группы маршрутов. В данном случае мы снабжаем все сначала префиксом `users.`, а затем — `comments.` (пример 3.17).

Пример 3.17. Префиксы имен для групп маршрутов

```
Route::name('users.')->prefix('users')->group(function () {
    Route::name('comments.')->prefix('comments')->group(function () {
        Route::get('{id}', function () {

        })->name('show');
    });
});
```

Подписанные маршруты

Многие приложения регулярно отправляют уведомления о разовых действиях (сброс пароля, принятие приглашения и т. д.) и предоставляют простые ссылки для их выполнения. Предположим, мы отправляем электронное письмо, предлагающее пользователю подтвердить согласие на рассылку.

Для этого есть три способа.

- ❑ Можно сделать этот URL-адрес публичным и надеяться, что никто не обнаружит ссылку для подтверждения и не модифицирует свою так, чтобы можно было выполнить согласие за кого-то другого.
- ❑ Можно поместить действие за аутентификацией, предоставить ссылку на действие и потребовать, чтобы пользователь прошел аутентификацию, если он еще этого не сделал (в данном случае это невозможно, поскольку получатели могут не являться пользователями приложения).
- ❑ «Подписать» ссылку, чтобы она однозначно подтверждала, что пользователь получил ее из вашего электронного письма, без необходимости аутентификации; это выглядит примерно так: `myapp.com/invitations/5816/yes?signature=030ab0ef6a8237bd86a8b8`.

5.9 Один из простейших способов реализации последнего варианта — использование нововведения версии Laravel 5.6.12 — так называемых *подписанных URL-адресов*, призванных упростить создание системы аутентификации подписи для отправки таких ссылок. Они состоят из обычной маршрутной ссылки с добавленной «подписью», которая подтверждает, что URL-адрес не был изменен с момента отправки (и, следовательно, никто не модифицировал адрес для доступа к чужой информации).

Подписание маршрута

Чтобы можно было создать подписанный URL-адрес для доступа к заданному маршруту, у него должно быть имя:

```
Route::get('invitations/{invitation}/{answer}', 'InvitationController')
    ->name('invitations');
```

Чтобы сгенерировать обычную ссылку на этот маршрут, можно использовать уже рассмотренный нами хелпер `route()`, а также фасад URL: `URL::route('invitations', ['invitation' => 12345, 'answer' => 'yes'])`. Для *подписанной* ссылки добавьте вместо этого метод `signedRoute()`. А если вам нужен подписанный маршрут с ограниченным сроком действия, поможет `temporarySignedRoute()`:

```
// Генерирование нормальной ссылки
URL::route('invitations', ['invitation' => 12345, 'answer' => 'yes']);

// Генерирование подписанной ссылки
URL::signedRoute('invitations', ['invitation' => 12345, 'answer' => 'yes']);

// Генерирование подписанной ссылки с ограниченным сроком действия (временной)
URL::temporarySignedRoute(
    'invitations',
    now()->addHours(4),
    ['invitation' => 12345, 'answer' => 'yes']
);
```



Использование хелпера now()

5 С версии 5.5 в Laravel есть хелпер `now()`, который выполняет то же, что и метод `Carbon::now()` — возвращает экземпляр объекта `Carbon` в его текущем виде. Если вы работаете с более ранней версией фреймворка, то используйте `Carbon::now()` там, где в этой книге применяется `now()`.

`Carbon` — это включенная в состав `Laravel` библиотека для работы с датой и временем.

Изменение маршрутов для разрешения подписанных ссылок

Сгенерировав подписанную ссылку на маршрут, необходимо также предотвратить доступ к нему без подписи. Самый простой способ — использовать `signed` (если его нет в массиве `$routeMiddleware` в файле `app/Http/Kernel.php`, он должен быть продублирован в `Illuminate\Routing\Middleware\ValidateSignature`):

```
Route::get('invitations/{invitation}/{answer}', 'InvitationController')
    ->name('invitations')
    ->middleware('signed');
```

При желании вместо `signed` можно выполнять проверку вручную с помощью метода `hasValidSignature()` объекта `Request`:

```
class InvitationController
{
    public function __invoke(Invitation $invitation, $answer, Request $request)
    {
        if (!$request->hasValidSignature()) {
            abort(403);
        }

        //
    }
}
```

Представления (views)

В некоторых из рассмотренных нами замыканий маршрутов присутствовали строки вида `return view('account')`. Что это?

В паттерне MVC (см. рис. 3.1) *представления* (или шаблоны) — это файлы, которые описывают, как должен выглядеть какой-либо конкретный вывод. У вас могут быть представления для JSON, XML или электронных писем, однако их большая часть в веб-фреймворке служит для вывода HTML-кода.

Laravel предоставляет «из коробки» два формата представлений — шаблоны на обычном языке PHP или Blade (см. главу 4). При этом используются разные имена: файл `about.php` будет отображаться движком PHP, а `about.blade.php` — движком Blade.



Три способа загрузить представление

Есть три разных способа вернуть представление. Пока можете ограничиться использованием хелпера `view()`, но если вы увидите в коде метод `View::make()`, то учтите, что он аналогичен внедрению `Illuminate\View\ViewFactory`.

После «загрузки» представления хелпером `view()` можно просто вернуть его (как в примере 3.18), что подходит для случая, когда в представлении не используются какие-либо переменные из контроллера.

Пример 3.18. Простое использование хелпера `view()`

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

Этот код находит представление в файле `resources/views/home.blade.php` или `resources/views/home.php`, загружает его содержимое и производит синтаксический разбор встроенного PHP-кода и управляющих структур, выдавая на выходе только вывод. После возвращения он передается по всему стеку ответов и возвращается пользователю.

Но если вам нужно передавать в представления переменные? Взгляните на пример 3.19.

Пример 3.19. Передача переменных в представления

```
Route::get('tasks', function () {
    return view('tasks.index')
        ->with('tasks', Task::all());
});
```


Это замыкание загружает представление `resources/views/tasks/index.blade.php` или `resources/views/tasks/index.php` и передает ему одну переменную с именем `tasks`, которая содержит результат метода `Task::all()`. `Task::all()` — это запрос к базе данных Eloquent, о котором вы узнаете из главы 5.

Прямой возврат простых маршрутов с помощью метода `Route::view()`

5.5 Поскольку маршрут часто просто возвращает представление без каких-либо пользовательских данных, в Laravel версии 5.5 и выше можно определить его как маршрут «представления», даже не передавая замыкание или ссылку на контроллер/метод, как показано в примере 3.20.

Пример 3.20. `Route::view()`

```
// Возвращает resources/views/welcome.blade.php
Route::view('/', 'welcome');

// Передает простые данные в Route::view()
Route::view('/', 'welcome', ['User' => 'Michael']);
```

Общий доступ представлений к переменным с использованием компоновщиков представлений

Иногда возникает необходимость раз за разом передавать одни и те же переменные. Так, иногда переменную нужно сделать доступной для всех представлений сайта, определенного класса или включенного подпредставления — например, для связанных с задачами или разделом заголовка.

Можно сделать определенные переменные доступными для каждого шаблона или только для конкретных шаблонов, как показано в следующем коде:

```
view()->share('variableName', 'variableValue');
```

Подробнее об этом написано в разделе «Компоновщики представлений и внедрение сервисов» на с. 98.

Контроллеры (controllers)

Хотя я уже несколько раз упоминал контроллеры, в большинстве рассмотренных примеров использовались замыкания маршрутов. В паттерне MVC контроллеры являются классами, сводящими в одно место логику одного или нескольких маршрутов. Они часто группируют схожие пути, особенно если структура приложения соответствует традиционной схеме CRUD; в таком случае контроллер может поддерживать весь набор действий, осуществляемых над определенным ресурсом.



Что такое CRUD

CRUD обозначает создание, чтение, обновление, удаление (create, read, update, delete) — четыре основные операции, которые обычно веб-приложения предоставляют для ресурса. Например, вы можете создать новое сообщение в блоге, прочитать, обновить или удалить его.

Несмотря на большой соблазн «втиснуть» в контроллеры всю логику приложения, лучше думать о них как о регулировщиках движения, которые направляют HTTP-запросы внутри вашего приложения. Поскольку существуют и другие пути — задачи сгон, вызовы Artisan из командной строки, очереди задач и т. д., — будет разумно не слишком полагаться на контроллеры в плане реализации поведения. Это означает, что основная их задача состоит в том, чтобы определить цель HTTP-запроса и передать его остальной части приложения.

Таким образом, создадим контроллер. Один из простейших способов — воспользоваться командой Artisan, поэтому выполните в командной строке следующую команду:

```
php artisan make:controller TasksController
```



Artisan и генераторы Artisan

Laravel поставляется в комплекте с инструментом командной строки под названием Artisan. Его можно использовать для ручного запуска миграций, создания пользователей и других записей базы данных, а также для выполнения множества иных ручных однократных задач.

В пространстве имен make Artisan предоставляет инструменты для генерирования каркасных файлов для различных системных. Именно поэтому мы можем выполнить команду `php artisan make:controller`.

Чтобы узнать больше об этой и других функциях Artisan, см. главу 8.

Это приведет к созданию нового файла с именем `TasksController.php` в папке `app/Http/Controllers`, содержимое которого показано в примере 3.21.

Пример 3.21. Сгенерированный по умолчанию контроллер

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TasksController extends Controller
{
    //
}
```

Измените этот файл, как показано в примере 3.22, создав новый публичный метод `index()`. Он будет просто возвращать некоторый текст.

Пример 3.22. Пример простого контроллера

```
<?php

namespace App\Http\Controllers;

class TasksController extends Controller
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

Затем уже известным нам способом следует подключить к этому контроллеру маршрут, как показано в примере 3.23.

Пример 3.23. Маршрут для простого контроллера

```
// routes/web.php
<?php

Route::get('/', 'TasksController@index');
```

Вот и все. Теперь при переходе по пути / будут выводиться слова Hello, World!.

ПРОСТРАНСТВО ИМЕН КОНТРОЛЛЕРА

В примере 3.23 мы ссылались на контроллер, полностью определенное имя класса которого выглядит как `App\Http\Controllers\TasksController`, используя при этом только название класса. Это объясняется тем, что мы можем опускать сегмент `App\Http\Controllers\` при обращении к контроллерам, поскольку Laravel по умолчанию настроен на их поиск в этом пространстве имен.

Это означает, что если полностью определенное имя класса контроллера выглядит как `App\Http\Controllers\API\ExercisesController`, то в определении маршрута его можно указать как `API\ExercisesController`.

Таким образом, наиболее типичный способ использования метода контроллера выглядит примерно так, как показано в примере 3.24, где он предоставляет ту же функциональность, что и замыкание маршрута в примере 3.19.

Пример 3.24. Пример типичного метода контроллера

```
// TasksController.php
...
public function index()
{
    return view('tasks.index')
        ->with('tasks', Task::all());
}
```

Так загружается представление `resources/views/tasks/index.blade.php` или `resources/views/tasks/index.php` и ему передается одна переменная с именем `tasks`, которая содержит результат метода `Task::all()` для выполнения Eloquent-запроса.



Генерирование контроллеров ресурсов

❗ Если вы выполняли команду `php artisan make:controller` в Laravel до версии 5.3, то, вероятно, ожидаете, что она будет автоматически генерировать такие методы, как `create()` и `update()`, для всех основных маршрутов ресурсов. Вы можете добиться такого же поведения в Laravel 5.3 и выше, указав флаг `--resource` при создании:

```
php artisan make:controller TasksController --resource
```

Получение ввода пользователя

Вторым наиболее распространенным действием является получение ввода от пользователя и осуществление над ним определенных манипуляций. Здесь используется несколько новых концепций, поэтому взглянем на пример кода и разберемся с незнакомыми элементами.

Во-первых, привяжем действия к нашему маршруту (пример 3.25).

Пример 3.25. Привязка основных действий формы

```
// routes/web.php
Route::get('tasks/create', 'TasksController@create');
Route::post('tasks', 'TasksController@store');
```

Обратите внимание, что мы привязываем GET к `tasks/create` (который показывает форму для создания новой задачи) и POST к `tasks/` (где наша форма будет размещать сообщение после создания новой задачи). Мы можем предположить, что `create()` в нашем контроллере просто показывает форму. Таким образом, взглянем на метод `store()` в примере 3.26.

Пример 3.26. Обычный метод ввода формы контроллера

```
// TasksController.php
...
public function store()
{
    Task::create(request()->only(['title', 'description']));

    return redirect('tasks');
}
```

В этом примере используются модели Eloquent и хелпер `redirect()`, о которых мы поговорим позже, а пока кратко рассмотрим, как мы здесь получаем данные.

Используем функцию `request()` для представления HTTP-запроса (подробнее об этом — позже) и его метод `only()` для извлечения из пользовательского ввода только полей заголовка `title` и описания `description`.

Затем мы передаем эти данные в метод `create()` нашей модели `Task`, который создает новый экземпляр объекта `Task` с `title`, содержащим переданный заголовок, и `description` с переданным описанием. Наконец, мы перенаправляем данные обратно на страницу со списком всех задач.

Здесь есть несколько уровней абстракции, которые мы рассмотрим чуть позже. Однако сразу же стоит отметить, что данные из метода `only()` поступают из того же пула данных, из которого берут информацию все обычно используемые методы объекта `Request`, в частности `all()` и `get()`. Такой набор данных — это все предоставленные пользователем материалы, будь то параметры запросов или значения `POST`. Таким образом, наш пользователь заполнил два поля на странице **Добавление задачи**: **Заголовок** (`title`) и **Описание** (`description`).

Что ж, разберемся с этой абстракцией. Метод `request()->only()` принимает ассоциативный массив имен полей ввода и возвращает их содержимое:

```
request()->only(['title', 'description']);  
// returns:  
[  
    'title' => 'Whatever title the user typed on the previous page',  
    'description' => 'Whatever description the user typed on the previous page',  
]
```

Метод `Task::create()` принимает ассоциативный массив и создает на его основе новую задачу:

```
Task::create([  
    'title' => 'Buy milk',  
    'description' => 'Remember to check the expiration date this time, Norbert!',  
]);
```

Объединение двух методов позволяет получить задачу, содержащую только предоставленные пользователем поля **Заголовок** и **Описание**.

Внедрение зависимостей в контроллеры

Фасады и глобальные хелперы фреймворка — это простой интерфейс для наиболее полезных классов в кодовой базе Laravel. Они позволяют получить информацию о текущем запросе и пользовательском вводе, сессии, кэшах и многом другом.

Если вы предпочитаете внедрять свои зависимости или хотите использовать сервис, у которого нет фасада или хелпера, то потребуется каким-то образом передавать экземпляры этих классов в контроллер.

Это наше первое знакомство с сервисным контейнером Laravel. Если вы незнакомы с данной концепцией, пока можете считать это одним из «фокусов» фреймворка или все-таки узнать подробнее о том, что она собой представляет, сразу перейдя к главе 11.

Все методы контроллеров (включая конструкторы) разрешаются из контейнера Laravel. Это означает, что автоматически внедряется все, что имеет подсказки типов, которые может разрешить контейнер.



Подсказки типов в PHP

Предоставление подсказки типа в PHP означает размещение имени класса или интерфейса перед переменной в сигнатуре метода:

```
public function __construct(Logger $logger) {}
```

Эти подсказки типа говорят PHP, что все переданное в метод должно иметь тип `Logger`, который может быть интерфейсом или классом.

В качестве примера посмотрим, как можно использовать экземпляр объекта `Request` вместо глобального хелпера. Для этого просто снабдите параметры метода подсказкой типа `Illuminate\Http\Request`, как показано в примере 3.27.

Пример 3.27. Внедрение метода контроллера путем предоставления подсказки типа

```
// TasksController.php
...
public function store(\Illuminate\Http\Request $request)
{
    Task::create($request->only(['title', 'description']));

    return redirect('tasks');
}
```

Таким образом, вы определили параметр, который нужно передавать в метод `store()`. Поскольку вы снабдили параметр типом подсказки и Laravel знает, как разрешить это имя класса, объект `Request` будет готов к использованию в вашем методе без каких-либо дополнительных усилий. Не требуется выполнять явное связывание или что-либо еще — объект уже в вашем распоряжении в виде переменной `$request`.

Как вы можете увидеть, сравнив примеры 3.26 и 3.27, хелпер `request()` и объект `Request` ведут себя одинаково.

Контроллеры ресурсов

Иногда именование методов в ваших контроллерах может быть самой сложной задачей. К счастью, у Laravel есть соглашения для всех маршрутов традиционного контроллера REST/CRUD (который называется контроллером ресурсов). Кроме того, он поставляется с готовым генератором и удобным определением маршрута, что позволяет вам привязать весь контроллер ресурсов за раз.

Чтобы увидеть методы, которые Laravel ожидает для контроллера ресурсов, сгенерируем новый контроллер из командной строки:

```
php artisan make:controller MySampleResourceController --resource
```

Теперь откройте файл `app/Http/Controllers/MySampleResourceController.php`. Вы увидите, что он уже содержит достаточно много методов. Кратко рассмотрим, что представляет собой каждый метод, на примере ресурса `Task`.

Методы контроллеров ресурсов Laravel

Помните приведенную ранее таблицу? В табл. 3.1 показаны HTTP-команды, URL-адреса, методы контроллера и имена по умолчанию, которые генерируются в контроллерах ресурсов Laravel.

Привязка контроллера ресурса

Итак, мы увидели, что это имена маршрутов в соответствии с принятым в Laravel соглашением и что мы можем легко создавать контроллеры ресурсов с методами для каждого из этих маршрутов по умолчанию. К счастью, вам не нужно создавать пути для каждого вручную, если в этом нет особой необходимости. Есть хитрость, называемая *привязкой контроллера ресурсов*. Посмотрите на пример 3.28.

Пример 3.28. Привязка контроллера ресурса

```
// routes/web.php
Route::resource('tasks', 'TasksController');
```

Этот код автоматически свяжет все маршруты из табл. 3.1 с соответствующими методами указанного контроллера. Они будут названы соответствующим образом; например, метод `index()` контроллера ресурса `tasks` будет называться `tasks.index`.



artisan route:list

Если вам станет интересно, какие маршруты доступны для вашего текущего приложения, для этого есть инструмент: из командной строки запустите команду `php artisan route:list`, и вы получите полный список (рис. 3.2).

```
mattstauffer at Cassim in ~/Sites/book-up-and-running
o php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	api/user		Closure	api,auth:api
	GET HEAD	dogs	dogs.index	App\Http\Controllers\DogsController@index	web
	POST	dogs	dogs.store	App\Http\Controllers\DogsController@store	web
	GET HEAD	dogs/create	dogs.create	App\Http\Controllers\DogsController@create	web
	GET HEAD	dogs/{dog}	dogs.show	App\Http\Controllers\DogsController@show	web
	PUT PATCH	dogs/{dog}	dogs.update	App\Http\Controllers\DogsController@update	web
	DELETE	dogs/{dog}	dogs.destroy	App\Http\Controllers\DogsController@destroy	web
	GET HEAD	dogs/{dog}/edit	dogs.edit	App\Http\Controllers\DogsController@edit	web

Рис. 3.2. artisan route:list

Контроллеры ресурсов API

Когда вы работаете с API RESTful, список возможных действий с ресурсом не совпадает со списком для контроллера ресурсов HTML. Например, вы можете отправить запрос POST в API для создания ресурса, но вы не можете там «показать форму создания».

5.6 В Laravel 5.6 появился новый способ генерирования *контроллера ресурсов API*, который имеет ту же структуру, лишь с тем отличием, что у него нет действий *создания и редактирования*. Мы можем сгенерировать контроллеры ресурсов API, передавая флаг `--api` при создании:

```
php artisan make:controller MySampleResourceController --api
```

Привязка контроллера ресурса API. Чтобы привязать контроллер ресурсов API, используйте метод `apiResource()` вместо метода `resource()`, как показано в примере 3.29.

Пример 3.29. Привязка контроллера ресурса API

```
// routes/web.php
Route::apiResource('tasks', 'TasksController');
```

Контроллеры одиночного действия

Иногда в приложениях требуется использовать контроллер для одного маршрута. Вы можете задаться вопросом именования. К счастью, можно связывать один маршрут с одним контроллером, не заботясь о том, как следует назвать его единственный метод.

В арсенале магических методов PHP есть метод `__invoke()`, позволяющий вызвать экземпляр класса и обратиться к нему как к функции. В Laravel можно применять

этот инструмент к *контроллерам одиночного действия* для сопоставления одного маршрута с одним контроллером (пример 3.30).

Пример 3.30. Использование метода `invoke()`

```
// \App\Http\Controllers\UpdateUserAvatar.php
public function __invoke(User $user)
{
    // Обновить изображение на аватаре пользователя
}

// routes/web.php
Route::post('users/{user}/update-avatar', 'UpdateUserAvatar');
```

Привязка модели маршрута

В одном из наиболее распространенных шаблонов маршрутизации первая строка любого метода контроллера пытается найти ресурс с заданным идентификатором, как в примере 3.31.

Пример 3.31. Получение ресурса для каждого маршрута

```
Route::get('conferences/{id}', function ($id) {
    $conference = Conference::findOrFail($id);
});
```

Laravel позволяет упростить реализацию шаблона за счет применения так называемой *привязки модели маршрута*. Вы можете определить, что конкретное имя параметра (например, `{conference}`) будет указывать локатору маршрутов, что он должен найти запись базы данных Eloquent с этим идентификатором, а затем передать ее в качестве параметра *вместо* того, чтобы просто передавать идентификатор.

Существует два вида привязки модели маршрута: неявная и пользовательская (или явная).

Неявная привязка модели маршрута

Самый простой способ использовать привязку модели маршрута — присвоить параметру маршрута какое-то уникальное имя для этой модели (например, назвать его `$conference` вместо `$id`), затем определить тип этого параметра в методе замыкания/контроллера и указать там то же имя переменной. Это проще показать, чем описать, поэтому взгляните на пример 3.32.

Пример 3.32. Использование неявной привязки модели маршрута

```
Route::get('conferences/{conference}', function (Conference $conference) {
    return view('conferences.show')->with('conference', $conference);
});
```

Поскольку параметр маршрута (`{conference}`) совпадает с параметром метода (`$conference`), а параметр метода указан с типом модели `Conference` (`Conference $conference`), Laravel видит это как привязку модели маршрута. Каждый раз при проходе этого пути приложение будет предполагать, что все переданное в URL-адресе вместо `{conference}` является идентификатором, который должен использоваться для поиска объекта `Conference`, а затем этот результирующий экземпляр модели будет передаваться вашему методу замыкания или контроллера.



Настройка ключа маршрута для модели Eloquent

Каждый раз, когда модель Eloquent просматривается посредством сегмента URL-адреса (обычно в силу применения привязки модели маршрута), по умолчанию поиск производится по столбцу модели Eloquent, являющемуся ее первичным ключом (по столбцу с идентификаторами).

Чтобы поиск на основе URL-адреса производился по другому столбцу модели Eloquent, добавьте в модель метод с именем `getRouteKeyName()`:

```
public function getRouteKeyName()
{
    return 'slug';
}
```

Теперь URL-адрес вида `conference/{conference}` будет ожидать запись из столбца `slug`, а не из столбца с идентификаторами и будет выполняться поиск в базе данных.

5.2 Неявная привязка модели маршрута была добавлена в Laravel 5.2, поэтому вы не сможете использовать эту возможность в версии 5.1.

Пользовательская привязка модели маршрута

Чтобы вручную настроить привязку модели маршрута, добавьте строку, подобную приведенной в примере 3.33, в метод `boot()` в `App\Providers\RouteServiceProvider`.

Пример 3.33. Добавление привязки модели маршрута

```
public function boot()
{
    // Позволяем родительскому методу boot() работать по-прежнему
    parent::boot();

    // Выполняем привязку
    Route::model('event', Conference::class);
}
```

Теперь вы указали, что всякий раз, когда у маршрута будет параметр в его определении с именем `{event}`, как показано в примере 3.34, локатор маршрута станет

возвращать экземпляр класса `Conference` с идентификатором этого параметра URL-адреса.

Пример 3.34. Использование явной привязки модели маршрута

```
Route::get('events/{event}', function (Conference $event) {  
    return view('events.show')->with('event', $event);  
});
```

Кэширование маршрутов

Если вы хотите сэкономить время во время загрузки, то можете использовать кэширование маршрутов. Одной из существенных составляющих начальной загрузки в Laravel, способной длиться от нескольких десятков до нескольких сотен миллисекунд, является синтаксический разбор файлов `routes/*`, и кэширование значительно ускоряет этот процесс.

Для этого необходимы все маршруты контроллера, перенаправления, представления и ресурсов (без замыканий маршрутов). Если ваше приложение не использует замыкания маршрутов, вы можете выполнить команду `php artisan route:cache`, и Laravel будет сериализовать результаты файлов `routes/*`. Для удаления кэша следует выполнить команду `php artisan route:clear`.

Однако есть недостаток: Laravel теперь будет сопоставлять маршруты с этим кэшированным файлом, а не с вашими `routes/*`. Можно вносить бесконечные изменения в свои файлы маршрутов, и они не вступят в силу, пока вы не выполните команду `route:cache` снова. Значит, придется повторять кэширование при каждом изменении, что иногда приводит к путанице.

Вместо этого я бы порекомендовал следующее: поскольку система управления версиями Git в любом случае по умолчанию игнорирует файл кэша маршрутов, используйте кэширование маршрутов только на эксплуатационном сервере и выполняйте команду `php artisan route:cache` при каждом развертывании нового кода (используя скрипт после развертывания Git, команду Forge или любую другую систему развертывания). Таким образом, у вас не возникнет трудностей с локальной разработкой, но ваша удаленная среда все равно выиграет от кэширования маршрутов.

Подмена метода формы

Иногда требуется вручную указать, какую HTTP-команду должна отправлять форма. HTML-формы позволяют использовать только `GET` или `POST`, поэтому, если вы хотите использовать другую команду, нужно будет указать это самостоятельно.

HTTP-команды в Laravel

Вы можете узнать, каким командам будет соответствовать маршрут при его определении, используя методы `Route::get()`, `Route::post()`, `Route::any()` и `Route::match()`. Можно выполнить сопоставление с помощью `Route::patch()`, `Route::put()` и `Route::delete()`.

Но как отправить запрос, отличный от GET, из браузера? Например, выбор HTTP-команды зависит от атрибута `method` HTML-формы: со значением GET она будет отправляться параметрами запроса и методом GET; с POST — телом сообщения и методом POST.

Фреймворки JavaScript позволяют легко отправлять другие запросы, такие как DELETE и PATCH. Но если вам нужно отослать HTML-формы в Laravel с использованием команд, отличных от GET или POST, придется применять *подмену метода формы*, что означает замещение HTTP-метода в HTML-форме.

Подмена HTTP-метода в HTML-формах

Чтобы сообщить Laravel, что отправляемая вами в настоящее время форма должна рассматриваться как нечто отличное от POST, добавьте скрытую переменную с именем `_method` и значением PUT, PATCH или DELETE. Фреймворк сопоставит и маршрутизирует формы этого отправления так, словно действительно был запрос с указанной командой.

Форма в примере 3.35 передает Laravel метод DELETE и потому будет сопоставлена с маршрутами, определяемыми `Route::delete()`, но не `Route::post()`.

Пример 3.35. Подмена метода формы

```
<form action="/tasks/5" method="POST">
  <input type="hidden" name="_method" value="DELETE">
  <!-- или: -->
  @method('DELETE')
</form>
```

Защита CSRF

Если вы уже пытались отправить форму в приложении Laravel, в том числе в примере 3.35, то, вероятно, столкнулись с ужасной ошибкой `TokenMismatchException`.

По умолчанию все маршруты в Laravel, кроме предназначенных только для чтения (то есть использующих команды GET, HEAD и OPTIONS), защищены от атак типа «подделка межсайтовых запросов» (Cross-Site Request Forgery, CSRF) посредством запрашивания токена в виде входного параметра с именем `_token`, передаваемого с каждым запросом. Этот токен генерируется в начале каждой сессии, и каждый

маршрут, не предназначенный только для чтения, сравнивает переданный параметр `_token` с токеном сессии.



Что такое CSRF

Подделка межсайтовых запросов — это когда один сайт притворяется другим. Цель злоумышленников — перехватить доступ пользователей к вашему сайту, отправляя формы со своего сайта на ваш через браузер аутентифицированного пользователя.

Лучший способ защиты от атак CSRF состоит в том, чтобы защитить все входящие маршруты — POST, DELETE и т. д. — с помощью токена, что Laravel делает по умолчанию.

У вас есть два варианта обойти эту ошибку CSRF. Первый и предпочтительный метод заключается в добавлении входного параметра `_token` к каждой отправляемой форме. В случае HTML-форм это легко; посмотрите на пример 3.36.

Пример 3.36. Токены CSRF

```
<form action="/tasks/5" method="POST">
  <?php echo csrf_field(); ?>
  <!-- или: -->
  <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>" />
  <!-- или: -->
  @csrf
</form>
```



Хелперы CSRF в Laravel до версии 5.6

Директива Blade `@csrf` недоступна в проектах с Laravel версий до 5.6. Вместо этого нужно использовать хелпер `csrf_field()`.

В приложениях JavaScript это сделать сложнее. Наиболее распространенное решение для сайтов, использующих JavaScript-фреймворки, — сохранение токена на каждой странице в теге `<meta>` следующего вида:

```
<meta name="csrf-token" content="<?php echo csrf_token(); ?>" id="token">
```

Хранение токена в теге `<meta>` позволяет легко привязать его к правильному HTTP-заголовку, что вы можете сделать один раз глобально для всех запросов из вашего фреймворка JavaScript, как в примере 3.37.

Пример 3.37. Глобальная привязка заголовка для CSRF

```
// В jQuery:
$.ajaxSetup({
  headers: {
```

```

        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});

```

```
// С Axios:
```

```

window.axios.defaults.headers.common['X-CSRF-TOKEN'] =
    document.head.querySelector('meta[name="csrf-token"]');

```

Laravel будет проверять X-CSRF-TOKEN в каждом запросе, и если там содержится правильный токен, защита CSRF будет помечена как выполненная.

Обратите внимание, что в этом примере не нужен синтаксис Vue для CSRF, если вы работаете с начальной загрузкой Vue по умолчанию в установке Laravel; фреймворк все сделает сам.



Связывание токенов CSRF с помощью Vue Resource

В проектах под управлением Laravel 5.3 и более ранних версий и Vue вы можете полагаться на библиотеку Vue Resource (<http://bit.ly/2UbVklZ>) для выполнения вызовов Ajax. Начальная загрузка токена CSRF во Vue Resource выглядит несколько иначе, чем в случае Laravel; примеры см. в документации по Vue Resource.

Перенаправления

До сих пор мы рассматривали лишь случай возвращения представления из метода контроллера или определения маршрута. Однако сообщить браузеру нужную модель поведения можно и путем возвращения ряда других структур.

Рассмотрим *перенаправления*. Вы уже видели некоторые в предыдущих примерах. Существует два распространенных способа их создания; здесь мы будем использовать глобальный хелпер `redirect()`, но подходит и фасад. Оба создают экземпляр `Illuminate\Http\RedirectResponse`, применяют к нему несколько методов для удобства, а затем возвращают его. Можно сделать это вручную, но придется выполнить немного больше работы самостоятельно. Взгляните на пример 3.38, где показано несколько способов возврата перенаправления.

Пример 3.38. Различные способы вернуть перенаправление

```

// Использование глобального хелпера
// для генерации ответа перенаправления
Route::get('redirect-with-helper', function () {
    return redirect()->to('login');
});

// Использование глобального хелпера с сокращенной формой
Route::get('redirect-with-helper-shortcut', function () {
    return redirect('login');
});

```

```
// Использование фасада для генерации ответа перенаправления
Route::get('redirect-with-facade', function () {
    return Redirect::to('login');
});

// Использование сокращенной формы Route::redirect в Laravel 5.5+
Route::redirect('redirect-by-route', 'login');
```

Обратите внимание, что хелпер `redirect()` предоставляет те же методы, что и фасад `Redirect`, но у него есть сокращенная форма. Если вы передаете параметры непосредственно хелперу вместо цепочки методов после него, это сокращенная форма для метода перенаправления `to()`.

Заметьте также, что (необязательный) третий параметр для хелпера маршрута `Route::redirect()` может быть кодом состояния (например, 302) для вашего перенаправления.

`redirect()->to()`

Подпись метода `to()` для перенаправлений выглядит следующим образом:

```
function to($to = null, $status = 302, $headers = [], $secure = null)
```

`$to` — допустимый внутренний путь, `$status` — статус HTTP (по умолчанию 302), `$headers` позволяет определить, какие HTTP-заголовки отправлять вместе с вашим перенаправлением, а `$secure` переопределяет выбор по умолчанию: `http` или `https` (который обычно устанавливается на основе вашего текущего URL-адреса запроса). В примере 3.39 показан вариант использования.

Пример 3.39. Перенаправление `redirect()->to()`

```
Route::get('redirect', function () {
    return redirect()->to('home');

    // Или то же самое с использованием сокращенной формы
    return redirect('home');
});
```

`redirect()->route()`

Метод `route()` аналогичен `to()`, но вместо указания конкретного пути мы ссылаемся на определенное имя маршрута (пример 3.40).

Пример 3.40. Перенаправление `redirect()->route()`

```
Route::get('redirect', function () {
    return redirect()->route('conferences.index');
});
```

Некоторые имена маршрутов требуют наличия параметров, но порядок их параметров немного отличается. У метода `route()` есть опциональный второй параметр маршрута:

```
function route($to = null, $parameters = [], $status = 302, $headers = [])
```

Это перенаправление можно использовать так, как показано в примере 3.41.

Пример 3.41. Перенаправление `redirect()->route()` с параметрами

```
Route::get('redirect', function () {
    return redirect()->route('conferences.show', ['conference' => 99]);
});
```

redirect()->back()

Благодаря встроенным возможностям реализации сессии Laravel ваше приложение всегда будет знать, какую страницу пользователь посетил ранее. Кроме того, реализуется возможность перенаправления `redirect()->back()`, возвращающего пользователя на ту страницу, с которой он пришел. Для этого также есть сокращенная форма `back()`.

Другие методы перенаправления

Сервис перенаправления предоставляет другие методы, используемые не так часто.

- ☐ `home()`. Перенаправляет на маршрут с именем `home`.
- ☐ `refresh()`. Перенаправляет на ту же страницу, на которой сейчас находится пользователь.
- ☐ `away()`. Позволяет перенаправить на внешний URL-адрес без проверки URL-адреса по умолчанию.
- ☐ `secure()`. Похож на `to()` с параметром `secure`, имеющим значение `true`.
- ☐ `action()`. Осуществляет привязку к контроллеру и методу одним из двух способов: в виде строки (`redirect()->action('MyController@myMethod')`) или кортежа (`redirect()->action([MyController::class, 'myMethod'])`).
- ☐ `guest()`. Используется внутренней системой аутентификации (обсуждается в главе 9); когда пользователь посещает неаутентифицированный маршрут, метод захватывает «предполагаемый» маршрут и затем перенаправляет пользователя (обычно на страницу авторизации).
- ☐ `intended()`. Также используется внутри системы аутентификации; после успешного распознавания метод получает «заданный» URL-адрес, сохраненный `guest()`, и перенаправляет туда пользователя.

redirect()->with()

Структурирован аналогично другим методам, которые вы можете вызывать с помощью `redirect()`. Функция `with()` отличается тем, что определяет не то, куда вы перенаправляете, а какие данные передаете вместе с перенаправлением. Когда вы перенаправляете пользователей на разные страницы, часто требуется передавать еще и определенные данные. Можно вручную перенести их в сессию, но в Laravel есть более удобные методы.

Чаще с помощью `with()` передается либо массив ключей и значений, либо один ключ и значение, как в примере 3.42. Так вы сохраняете данные `with()` в сессии только для следующей загрузки страницы.

Пример 3.42. Перенаправление с данными

```
Route::get('redirect-with-key-value', function () {
    return redirect('dashboard')
        ->with('error', true);
});

Route::get('redirect-with-array', function () {
    return redirect('dashboard')
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```



Цепочка методов на перенаправлениях

Как и во многих других фасадах, большинство вызовов фасада `Redirect` может принимать текущие (fluent) цепочки методов, например вызовы `with()` в примере 3.42. Подробнее о текучести вы узнаете из врезки «Что такое текучий интерфейс» на с. 125.

Вы также можете использовать метод `withInput()`, как показано в примере 3.43, для перенаправления с миганием поля ввода. Это наиболее распространено в случае ошибки проверки, когда вы хотите отправить пользователя обратно к форме, которую он только что заполнил.

Пример 3.43. Перенаправление с вводом формы

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

Route::post('form', function () {
    return redirect('form')
        ->withInput()
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

Самый простой способ получить мигающий ввод, переданный с помощью `withInput()`, — воспользоваться хелпером `old()`, который можно применять для получения всего старого ввода (`old()`) или только значения для определенного ключа, как показано в следующем примере, со вторым параметром в качестве значения по умолчанию, если нет старого. Обычно встречается в представлениях, позволяющих использовать этот HTML-код для таких представлений, как «создать» и «редактировать», для этой формы:

```
<input name="username" value="<%=  
  old('username', 'Default username instructions here');  
%>">
```

В контексте проверки есть также полезный метод для передачи ошибок вместе с ответом на перенаправление — `withErrors()`. Вы можете отослать его любому обработчику ошибок, который может быть строкой ошибок, массивом ошибок или чаще всего экземпляром `Validator Illuminate` (см. главу 10). В примере 3.44 показано, как его использовать.

Пример 3.44. Перенаправление с ошибками

```
Route::post('form', function (Illuminate\Http\Request $request) {  
    $validator = Validator::make($request->all(), $this->validationRules);  
  
    if ($validator->fails()) {  
        return back()  
            ->withErrors($validator)  
            ->withInput();  
    }  
});
```

Метод `withErrors()` автоматически делит переменную `$errors` с представлениями страницы, на которую она перенаправляется, чтобы вы могли обрабатывать ее так, как вам хочется.



Метод `validate()` по запросам

Не нравится, как выглядит код в примере 3.44? Существует простой и мощный инструмент, который позволит его легко оптимизировать (см. подраздел «Метод `validate()` объекта `Request`» на с. 206).

Отмена запроса

Помимо возврата представлений и перенаправлений, наиболее распространенным способом выхода из маршрута является отмена. Существует несколько глобально доступных методов (`abort()`, `abort_if()` и `abort_unless()`), которые опционально принимают в качестве параметров коды состояния HTTP, сообщение и массив заголовков.

Как показано в примере 3.45, методы `abort_if()` и `abort_unless()` принимают первый параметр, который проверяется на достоверность, и выполняют отмену в зависимости от результата.

Пример 3.45. 403 — запрещенные отмены

```
Route::post('something-you-cant-do', function (Illuminate\Http\Request $request) {
    abort(403, 'You cannot do that!');
    abort_unless($request->has('magicToken'), 403);
    abort_if($request->user()->isBanned, 403);
});
```

Пользовательские ответы

Существует еще несколько доступных вариантов для возврата, поэтому рассмотрим наиболее распространенные ответы после представлений, перенаправлений и отмен. Вы можете запускать эти методы в хелпере `response()` или фасаде `Response`.

`response()->make()`

Если вы хотите создать собственный HTTP-ответ, передайте свои данные в первый параметр `response()->make()`: например, `return response()->make(Hello, World!)`. Второй — код состояния HTTP, а третий — ваши заголовки.

`response()->json()` и `->jsonp()`

Чтобы создать собственный HTTP-ответ в JSON-коде, передайте содержимое с поддержкой JSON (массивы, коллекции или что-либо еще) методу `json()`: `return response()->json(User::all())`. Это похоже на `make()`, за исключением того, что `+json_encode+s` добавляет ваш контент и устанавливает соответствующие заголовки.

`response()->download()`, `->streamDownload()` и `->file()`

Чтобы отправить файл конечному пользователю для загрузки, передайте в `download()` экземпляр `SplFileInfo` или строковое имя файла с необязательным вторым параметром имени конечного файла загрузки. Например, `return response()->download('file501751.pdf', 'myFile.pdf')` отправляет файл `file501751.pdf` и переименовывает его при отправке в `myFile.pdf`.

Чтобы отобразить тот же файл в браузере (если это PDF-файл, изображение или что-то еще, что может обрабатывать браузер), используйте взамен `response()->file()`, который принимает те же параметры, что и `response()->download()`.

Если вы хотите сделать доступным для загрузки некоторый контент из внешнего сервиса, не записывая его непосредственно на диск вашего сервера, то можете

выполнить потоковую загрузку с помощью `response()->streamDownload()`. Этот метод ожидает в качестве параметров замыкание, которое повторяет строку, имя файла и — не обязательно — массив заголовков (пример 3.46).

Пример 3.46. Потоковая загрузка с внешних серверов

```
return response()->streamDownload(function () {
    echo DocumentService::file('myFile')->getContent();
}, 'myFile.pdf');
```

Тестирование

В некоторых других сообществах распространен метод модульного тестирования контроллера, но в Laravel (и большей части сообщества PHP) обычно выполняется *тестирование приложений* для проверки функциональности маршрутов.

Например, чтобы убедиться, что маршрут POST работает правильно, мы можем написать такой тест, как в примере 3.47.

Пример 3.47. Написание простого теста маршрута POST

```
// tests/Feature/AssignmentTest.php
public function test_post_creates_new_assignment()
{
    $this->post('/assignments', [
        'title' => 'My great assignment',
    ]);

    $this->assertDatabaseHas('assignments', [
        'title' => 'My great assignment',
    ]);
}
```

Мы вызывали напрямую методы контроллера? Нет. Но мы убедились, что цель этого маршрута достигнута: получение POST и сохранение его важной информации в базе данных.

Вы также можете использовать аналогичный синтаксис для посещения маршрута и проверки, что определенный текст отображается на странице или что нажатие конкретных кнопок приводит к выполнению определенных операций (пример 3.48).

Пример 3.48. Написание простого теста маршрута GET

```
// AssignmentTest.php
public function test_list_page_shows_all_assignments()
{
    $assignment = Assignment::create([
        'title' => 'My great assignment',
    ]);

    $this->get('/assignments')
        ->assertSee('My great assignment');
}
```



Различные названия для методов тестирования до Laravel 5.4

В проектах под управлением Laravel до версии 5.4 метод `assertDatabaseHas()` должен быть заменен на `seeInDatabase()`, а `get()` и `assertSee()` — на `visit()` и `see()`.

Резюме

Маршруты Laravel определены в файлах `routes/web.php` и `routes/api.php`. Можно задать ожидаемый путь каждого маршрута, статические сегменты, параметры, то, каким HTTP-командам доступен маршрут и как это разрешить. Вы также можете добавить промежуточное ПО к маршрутам, сгруппировать их и назвать.

То, что возвращается из метода замыкания маршрута или метода контроллера, определяет, как Laravel отвечает пользователю. Строка или представление выдаются пользователю, другие виды данных преобразуются в JSON и представляются пользователю, а перенаправление вызывает перенаправление.

Laravel предоставляет ряд инструментов и функций для упрощения общих задач и структур, связанных с маршрутизацией: контроллеры ресурсов, привязку модели маршрута и подмену метода формы.

4

Движок шаблонов Blade

PHP в качестве языка шаблонов функционирует относительно хорошо. Но у него есть свои недостатки, поэтому нельзя использовать `<?php inline` повсеместно. В большинстве современных фреймворков есть свой язык шаблонов.

Laravel предлагает собственный движок шаблонов *Blade*, созданный на основе движка .NET Razor. Он обладает лаконичным синтаксисом, довольно понятен, сопровождается мощной и интуитивно понятной моделью наследования и легкой расширяемостью.

Быстро ознакомиться с тем, как выглядит Blade, можно на примере 4.1.

Пример 4.1. Примеры Blade

```
<h1>{{ $group->title }}</h1>
{!! $group->heroImageHtml() !!}

@forelse ($users as $user)
    • {{ $user->first_name }} {{ $user->last_name }}<br>
@empty
    No users in this group.
@endforelse
```

Как видно, в коде Blade используются фигурные скобки для Echo и соглашение, в котором его пользовательские теги, называемые «директивами», имеют префикс `@`. Вы будете применять директивы для всех своих структур управления, а также для наследования и любых пользовательских функций, которые хотите добавить.

Синтаксис Blade чистый и лаконичный, поэтому работать с ним проще и приятнее, чем с альтернативами. Но в тот момент, когда понадобится что-нибудь сложное в ваших шаблонах — вложенное наследование, сложные условия или рекурсия, — движок проявляет себя с лучшей стороны. Как и лучшие компоненты Laravel, тяжелые требования к приложениям он упрощает и делает доступными.

Кроме того, поскольку весь синтаксис Blade скомпилирован в обычный код PHP, а затем кэширован, он быстр и позволяет по желанию использовать нативный PHP в ваших файлах этого движка. Тем не менее я бы рекомендовал избегать применения PHP, когда это вообще возможно, — обычно, если нужно сделать то, что невозможно с Blade или его пользовательской директивой, это не относится к шаблону.



Использование Twig с Laravel

В отличие от многих других фреймворков, основанных на Symfony, Laravel по умолчанию не использует Twig. Но если очень хочется, есть пакет Twig Bridge (<http://bit.ly/2U8dFt0>), который позволяет легко применять его вместо Blade.

Отображение данных

Как вы можете видеть в примере 4.1, скобки `{{ и }}` используются для обертки PHP-кода, который вы хотели бы отобразить. Код `{{ $variable }}` действует подобно `<?= $variable ?>` в простом PHP.

Однако есть отличие: Blade по умолчанию экранирует все отображения PHP-функцией `htmlspecialchars()` для защиты ваших пользователей от вставки вредоносных сценариев. Это означает, что `{{ $variable }}` функционально эквивалентно `<?= htmlspecialchars($variable) ?>`. Если вы не хотите экранировать вывод, используйте `{!! и !!}` вместо этого.

СКОБКИ `{{ и }}` ПРИ ИСПОЛЬЗОВАНИИ ФРОНТЕНД-ФРЕЙМВОРКА ШАБЛОНИЗАЦИИ

Вы могли заметить, что синтаксис отображения для Blade (`{{ }}`) аналогичен таковым для многих фронтенд-фреймворков. Как же Laravel узнает, используете вы Blade или Handlebars?

Blade игнорирует все `{{` с предваряющим знаком `@`. Таким образом, он проанализирует первый из следующих примеров, но второй будет выведен полностью:

```
// Распознается как Blade; значение $bladeVariable
// отображается в представлении
{{ $bladeVariable }}
```

```
// @ удаляется и "{{ handlebarsVariable }}" полностью
// отображается в представлении
@{{ handlebarsVariable }}
```

Вы также можете обернуть любые большие части содержимого сценария директивой `@verbatim` (<http://bit.ly/2OnrPRP>).

Управляющие структуры

Большинство управляющих структур в Blade будут знакомы. Многие напрямую дублируют имя и структуру такого же тега в PHP.

Есть несколько хелперов для удобства, но в целом структуры управления выглядят чище, чем в PHP.

Условные конструкции

Рассмотрим логические структуры управления.

@if

Выражение `@if ($condition)` в Blade компилируется в `<?php if ($condition): ?>`. `@else`, `@elseif` и `@endif` — с точно таким же стилем синтаксиса в PHP. Взгляните на пример 4.2.

Пример 4.2. `@if`, `@else`, `@elseif` и `@endif`

```
@if (count($talks) === 1)
    There is one talk at this time period.
}elseif (count($talks) === 0)
    There are no talks at this time period.
@else
    There are {{ count($talks) }} talks at this time period.
@endif
```

Как и в случае с собственными условными выражениями PHP, вы можете смешивать и комбинировать их так, как вам удобно. У них нет особой логики; есть анализатор с поиском в форме `@if($condition)` и заменой соответствующим кодом PHP.

@unless и @endunless

`@unless`, с другой стороны, — это новый синтаксис, который не имеет прямого эквивалента в PHP. Это противоположность `@if`. `@unless($condition)` совпадает с `<?php if (! $condition)`. Вы можете увидеть это на примере 4.3.

Пример 4.3. `@unless` и `@endunless`

```
@unless ($user->hasPaid())
    You can complete your payment by switching to the payment tab.
@endunless
```


Циклы

Далее рассмотрим циклы.

@for, @foreach и @while

@for, @foreach и @while работают в Blade так же, как и в PHP (примеры 4.4–4.6).

Пример 4.4. @for и @endfor

```
@for ($i = 0; $i < $talk->slotsCount(); $i++)  
    The number is {{ $i }}<br>  
@endfor
```

Пример 4.5. @foreach и @endforeach

```
@foreach ($talks as $talk)  
    • {{ $talk->title }} ({{ $talk->length }} minutes)<br>  
@endforeach
```

Пример 4.6. @while и @endwhile

```
@while ($item = array_pop($items))  
    {{ $item->orSomething() }}<br>  
@endwhile
```

@forelse и @endforelse

@forelse — это @foreach, который может быть выполнен, даже если перебираемый вами объект пуст. Мы видели это в действии в начале главы. Пример 4.7 показывает другой вариант.

Пример 4.7. @forelse

```
@forelse ($talks as $talk)  
    • {{ $talk->title }} ({{ $talk->length }} minutes)<br>  
@empty  
    No talks this day.  
@endforelse
```

ПЕРЕМЕННАЯ \$loop В ДИРЕКТИВАХ @FOREACH И @FORELSE



Директивы @foreach и @forelse (представленные в Laravel 5.3) добавляют переменную \$loop, недоступную в циклах foreach PHP. При использовании внутри цикла @foreach или @forelse она будет возвращать объект stdClass со следующими свойствами.

- **index** — отсчитанный от 0 индекс текущего элемента в цикле; 0 — «первый элемент».

- **iteration** — отсчитанный от 1 индекс текущего элемента в цикле; 1 — «первый элемент».
- **remaining** — количество элементов, оставшихся в цикле.
- **count** — количество элементов в цикле.
- **first** — логическое значение, указывающее, является ли данный элемент первым элементом в цикле.
- **last** — логическое значение, указывающее, является ли данный элемент последним элементом в цикле.
- **depth** — сколько «уровней» в этом цикле: 1 для цикла, 2 для цикла внутри цикла и т. д.
- **parent** — ссылка на переменную `$loop` для элемента родительского цикла, если этот цикл находится в другом цикле `@foreach`; иначе `null`.

Вот пример того, как это работает:

```
<ul>
@foreach ($pages as $page)
    <li>{{ $loop->iteration }}: {{ $page->title }}
        @if ($page->hasChildren())
            <ul>
                @foreach ($page->children() as $child)
                    <li>{{ $loop->parent->iteration }}
                        .{{ $loop->iteration }}:
                        {{ $child->title }}</li>
                @endforeach
            </ul>
        @endif
    </li>
@endforeach
</ul>
```

Наследование шаблонов

Blade предоставляет структуру для наследования шаблонов, которая позволяет представлениям расширять, изменять и включать в себя другие представления.

Посмотрим, как наследование структурируется с Blade.

Определение разделов страницы с помощью директив `@section/@show` и `@yield`

Начнем с макета Blade верхнего уровня, как в примере 4.8. Это определение универсальной обертки страницы, в которую мы позже поместим специфичный для страницы контент.

Пример 4.8. Структура Blade

```
<!-- resources/views/layouts/master.blade.php -->
<html>
  <head>
    <title>My Site | @yield('title', 'Home Page')</title>
  </head>
  <body>
    <div class="container">
      @yield('content')
    </div>
    @section('footerScripts')
      <script src="app.js"></script>
    @show
  </body>
</html>
```

Это напоминает обычную HTML-страницу, но вы можете видеть `yield` в двух местах (`title` и `content`), и мы определили `section` в третьем (`footerScripts`). Здесь у нас есть три директивы Blade: только `@yield('content')`, `@yield('title', 'Home Page')` с заданным по умолчанию значением и `@section/@show` с реальным содержимым в нем.

Хотя они выглядят немного по-разному, *но функционируют, по существу, одинаково*. Все три определяют, что есть раздел с заданным именем (первый параметр), который может быть расширен позже, и что делать, если раздел не был расширен. Они делают это либо строкой возврата (`'Home Page'`), либо без возврата (просто не будет отображаться ничего, если директива не расширена), либо с возвратом всего блока (в данном случае `<script src="app.js"></script>`).

В чем разница? У `@yield('content')` нет контента по умолчанию. Кроме того, в `@yield('title')` оно будет отображаться, *только* если директива не расширяется. В обратном случае ее дочерние разделы не будут иметь программного доступа к значению по умолчанию. `@section/@show` одновременно определяет значение по умолчанию и делает так, чтобы его содержимое было доступно его дочерним элементам через `@parent`.

Если у вас есть такой родительский макет, вы можете расширить его в новом файле шаблона, как в примере 4.9.

Пример 4.9. Расширение макета Blade

```
<!-- resources/views/dashboard.blade.php -->
@extends('layouts.master')

@section('title', 'Dashboard')

@section('content')
  Welcome to your application dashboard!
@endsection

@section('footerScripts')
  @parent
  <script src="dashboard.js"></script>
@endsection
```



@show по сравнению с @endsection

Возможно, вы заметили, что в примере 4.8 используется `@section/@show`, а в примере 4.9 — `@section/@endsection`. В чем разница?

Применяйте `@show`, когда вы определяете место для раздела в родительском шаблоне. А `@endsection`, если вы устанавливаете содержимое шаблона в дочернем шаблоне.

Это дочернее представление позволяет нам показать несколько новых концепций наследования Blade.

@extends

В примере 4.9 с помощью `@extends('layouts.master')` мы определяем, что это представление не должно отображаться само по себе, а вместо этого *расширяет* другое представление. Это означает, что его роль заключается в установке содержания различных разделов, но не в работе в одиночку. Больше похоже на серию блоков контента, чем на страницу HTML. Строка также определяет, что представление, которое она расширяет, находится по адресу `resources/views/layouts/master.blade.php`.

Каждый файл должен расширять только один другой файл, и вызов `@extends` обязан быть первой строкой файла.

@section и @endsection

С помощью `@section('title', 'Dashboard')` мы предоставляем наш контент для первого раздела, `title`. Поскольку содержимое очень короткое, вместо `@section` и `@endsection` мы используем сокращенную форму. Это позволяет передавать содержимое как второй параметр `@section`, а затем двигаться дальше. Если вас немного сбивает с толку `@section` без `@endsection`, можно использовать обычный синтаксис.

С `@section('content')` и далее мы используем обычный синтаксис для определения содержимого раздела `content`. Пока мы вставим небольшое приветствие. Однако заметьте: когда вы применяете `@section` в дочернем представлении, вы заканчиваете его `@endsection` (или его псевдонимом `@stop`) вместо `@show`, который зарезервирован для определения разделов в родительских представлениях.

@parent

С `@section('footerScripts')` и далее мы используем обычный синтаксис для определения содержимого раздела `footerScripts`.

Помните, что мы фактически определили этот контент (или по крайней мере его «значение по умолчанию») уже в главном макете. Так что на этот раз у нас есть два

варианта: либо *перезаписать* содержимое из родительского представления, либо *добавить* к нему.

Вы можете видеть, что у нас есть возможность включить содержимое из родительского представления с помощью директивы `@parent` внутри этого раздела. В противном случае содержимое раздела будет полностью переписано всем тем, что определено в родителе этого раздела.

Включение составляющих представления

Теперь, когда мы разобрались с основами наследования, можно использовать еще несколько уловок.

@include

Что, если мы находимся в одном представлении и хотим использовать другое? Возможно, есть кнопка регистрации, которую желательно повторно добавить по всему сайту. И, может быть, хочется подбирать текст кнопки каждый раз, когда ее используем. Посмотрите на пример 4.10.

Пример 4.10. Включение составляющих представления с `@include`

```
<!-- resources/views/home.blade.php -->
<div class="content" data-page-name="{{ $pageName }}">
    <p>Here's why you should sign up for our app: <strong>It's Great.</strong></p>

    @include('sign-up-button', ['text' => 'See just how great it is'])
</div>

<!-- resources/views/sign-up-button.blade.php -->
<a class="button button--callout" data-page-name="{{ $pageName }}">
    <i class="exclamation-icon"></i> {{ $text }}
</a>
```

`@include` подтягивает составляющую и передает в нее данные (не обязательно). Обратите внимание, что вы можете не только *явно* передавать данные для включения через второй параметр `@include`, но и ссылаться на любые переменные во включенном файле, которые доступны для включаемого представления (в этом примере `$pageName`). Вы можете делать все, что хотите, но я бы рекомендовал для ясности всегда передавать каждую переменную, которую собираетесь применять.

Вы также можете использовать директивы `@includeIf`, `@includeWhen` и `@includeFirst`, как показано в примере 4.11.

Пример 4.11. Включение представлений по условиям

```
{{-- Включить представление, если оно существует --}}
@includeIf('sidebars.admin', ['some' => 'data'])
```

```
{{-- Включить представление, если переданная переменная равна true --}}
@includeWhen($user->isAdmin(), 'sidebars.admin', ['some' => 'data'])

{{-- Включить первое представление из данного массива представлений --}}
@includeFirst(['customs.header', 'header'], ['some' => 'data'])
```

@each

Вы можете представить себе обстоятельства, когда вам нужно перебрать массив, коллекцию и `@include` часть для каждого элемента. Для этого есть директива `@each`.

Скажем, у нас есть боковая модульная панель и мы хотим включить несколько модулей со своим названием. Посмотрите на пример 4.12.

Пример 4.12. Использование составляющих представления в цикле с `@each`

```
<!-- resources/views/sidebar.blade.php -->
<div class="sidebar">
    @each('partials.module', $modules, 'module', 'partials.empty-module')
</div>

<!-- resources/views/partials/module.blade.php -->
<div class="sidebar-module">
    <h1>{{ $module->title }}</h1>
</div>

<!-- resources/views/partials/empty-module.blade.php -->
<div class="sidebar-module">
    No modules :(
</div>
```

Рассмотрим синтаксис `@each`. Первый параметр — это имя составляющей представления. Второй — массив или коллекция для итерации. Третье — название переменной, под которым каждый элемент (в данном случае каждый элемент в массиве `$modules`) будет передан представлению. И необязательный четвертый параметр — это представление, показывающее, являются ли массив или коллекция пустыми (или при желании можно передать здесь строку, которая будет использоваться в качестве вашего шаблона).

Использование стеков

5.2 Общий шаблон, возможно, сложен для управления с помощью базового Blade — когда каждому представлению в иерархии Blade необходимо добавить что-то в определенный раздел, — почти как при добавлении записи в массив.

Наиболее распространенная ситуация — когда определенные страницы (а иногда и в более широком смысле определенные разделы сайта) имеют конкретные

уникальные файлы CSS и JavaScript, которые им нужно загрузить. Представьте, что у вас есть «глобальный» CSS-файл для всего сайта, CSS-файл раздела вакансий и CSS-файл страницы «Устроиться на работу».

Стеки Blade созданы именно для такого. В родительском шаблоне определите стек-заполнитель. Затем в каждом дочернем шаблоне вы можете туда «вытаскивать» записи с помощью `@push/@endpush`, который добавляет их в конец стека в конечном изображении. Вы также можете использовать `@prepend/@endprepend`, чтобы добавить их в начало. Пример 4.13 иллюстрирует это.

Пример 4.13. Использование стеков Blade

```
<!-- resources/views/layouts/app.blade.php -->
<html>
<head><!-- шапка --></head>
<body>
  <!-- остальная часть страницы -->
  <script src="/css/global.css"></script>
  <!-- заполнитель, куда будет помещен контент из стека -->
  @stack('scripts')
</body>
</html>

<!-- resources/views/jobs.blade.php -->
@extends('layouts.app')

@push('scripts')
  <!-- вытаскиваем что-нибудь в конец стека -->
  <script src="/css/jobs.css"></script>
@endpush

<!-- resources/views/jobs/apply.blade.php -->
@extends('jobs')

@prepend('scripts')
  <!-- вытаскиваем что-нибудь в начало стека -->
  <script src="/css/jobs--apply.css"></script>
@endprepend
```

Это приводит к следующему результату:

```
<html>
<head><!-- шапка --></head>
<body>
  <!-- остальная часть страницы -->
  <script src="/css/global.css"></script>
  <!-- заполнитель, куда будет помещен контент из стека -->
  <script src="/css/jobs--apply.css"></script>
  <script src="/css/jobs.css"></script>
</body>
</html>
```

Использование компонентов и слотов

5.4 Laravel предлагает другой шаблон для включения контента между представлениями, который был представлен в 5.4: *компоненты* и *слоты*. Первые наиболее эффективны в контексте, когда вы используете составляющие представления и передаете в них большие порции контента в качестве переменных. Посмотрите на пример 4.14 для иллюстрации модели или всплывающей подсказки, которая может предупредить пользователя при ошибке или другом действии.

Пример 4.14. Модальное окно — неудачный пример составляющей представления

```
<!-- resources/views/partials/modal.blade.php -->
<div class="modal">
    <div>{{ $content }}</div>
    <div class="close button etc">...</div>
</div>

<!-- в другом шаблоне -->
@include('partials.modal', [
    'body' => '<p>The password you have provided is not valid. Here are the rules
for valid passwords: [...]</p><p><a href="#">...</a></p>'
])
```

Это слишком много для такой переменной и идеально подходит для компонента.

Компоненты со слотами — это составляющие представлений, которые разработаны, чтобы включать большие порции («слоты») для получения содержимого из включаемого шаблона. В примере 4.15 показано, как перепроектировать код из примера 4.14 с использованием компонентов и слотов.

Пример 4.15. Модальное окно как более подходящий компонент со слотами

```
<!-- resources/views/partials/modal.blade.php -->
<div class="modal">
    <div>{{ $slot }}</div>
    <div class="close button etc">...</div>
</div>

<!-- в другом шаблоне -->
@component('partials.modal')
    <p>The password you have provided is not valid.
    Here are the rules for valid passwords: [...]</p>

    <p><a href="#">...</a></p>
@endcomponent
```

Как вы можете видеть в примере 4.15, директива `@component` позволяет извлекать наш HTML-код из сжатой строки переменной и возвращает в пространство шаблона. Переменная `$slot` в шаблоне нашего компонента получает любой контент, передаваемый в `@component`.

Составные слоты

Метод, который мы использовали в примере 4.15, называется слотом «по умолчанию»; все, что вы передаете между `@component` и `@endcomponent`, передается переменной `$slot`. Но вы также можете иметь больше, чем просто слот по умолчанию. Представим модальное окно с заголовком, как в примере 4.16.

Пример 4.16. Составляющая модального представления с двумя переменными

```
<!-- resources/views/partials/modal.blade.php -->
<div class="modal">
    <div class="modal-header">{{ $title }}</div>
    <div>{{ $slot }}</div>
    <div class="close button etc">...</div>
</div>
```

Вы можете использовать директиву `@slot` в своих вызовах `@component` для передачи содержимого в слоты, отличные от заданного по умолчанию, как показано в примере 4.17.

Пример 4.17. Передача более одного слота компоненту

```
@component('partials.modal')
    @slot('title')
        Password validation failure
    @endslot

    <p>The password you have provided is not valid.
    Here are the rules for valid passwords: [...]</p>

    <p><a href="#">...</a></p>
@endcomponent
```

И если в вашем представлении есть другие переменные, которые бессмысленны в качестве слота, все равно можно передать массив содержимого в качестве второго параметра в `@component` так же, как с `@include`. Взгляните на пример 4.18.

Пример 4.18. Передача данных в компонент без слотов

```
@component('partials.modal', ['class' => 'danger'])
    ...
@endcomponent
```

Именованное компонента как директивы

Есть хитрый трюк, чтобы еще упростить компоненты для вызова: псевдонимы. Просто вызовите `Blade::component()` на фасаде `Blade` — наиболее распространенным является метод `AppServiceProvider boot()` — и передайте ему сначала местоположение компонента, а затем имя желаемой директивы, как показано в примере 4.19.

Пример 4.19. Псевдонимизация компонента как директивы

```
// AppServiceProvider@boot
Blade::component('partials.modal', 'modal');

<!-- в шаблоне -->
@modal
    Modal content here
@endmodal
```

**Импорт фасадов**

Мы впервые работаем с фасадом внутри класса с пространством имен. Подробнее рассмотрим это позже. Просто знайте, что при использовании фасадов внутри классов с пространством имен (а в последних версиях Laravel это большинство классов) может выдаваться сообщение о том, что нужный фасад не найден. Это объясняется тем, что фасады — обычные классы с обычными пространствами имен. Laravel позволяет обойти эту проблему, сделав нужный фасад доступным из корневого пространства имен.

Так, в примере 4.19 мы могли бы импортировать фасад `Illuminate\Support\Facades\Blade` в начале файла.

Компоновщики представлений и внедрение сервисов

Как мы рассмотрели в главе 3, передать данные в наши представления из определения маршрута просто (пример 4.20).

Пример 4.20. Напоминание о том, как передавать данные в представления

```
Route::get('passing-data-to-views', function () {
    return view('dashboard')
        ->with('key', 'value');
});
```

Возможны случаи, когда вы снова и снова передаете одни и те же данные нескольким представлениям. Или можно использовать составляющую заголовка, или что-то подобное, что требует некоторых данных. Придется ли вам передавать эти данные из каждого определения маршрута, которое может когда-либо загружать эту составляющую заголовка?

Привязка данных к представлениям с использованием компоновщиков представлений

К счастью, есть более простой способ использовать компоновщик представлений, позволяющий определить, что каждый раз, когда загружается конкретное представ-

ление, ему должны передаться некоторые данные — без необходимости передавать их явно из определения маршрута.

Допустим, у вас есть боковая панель на каждой странице, которая определена в составляющей с именем `partials.sidebar` (`resources/views/partials/sidebar.blade.php`). Она показывает список последних семи сообщений, опубликованных на вашем сайте. Если он находится на каждой странице, каждое определение маршрута обычно захватывает и передает этот список, как в примере 4.21.

Пример 4.21. Передача данных боковой панели из каждого маршрута

```
Route::get('home', function () {
    return view('home')
        ->with('posts', Post::recent());
});

Route::get('about', function () {
    return view('about')
        ->with('posts', Post::recent());
});
```

Скоро это начинает раздражать. Поэтому мы собираемся использовать компоновщики представлений, чтобы «поделиться» этой переменной с заданным набором представлений. Мы можем сделать это несколькими способами; начнем с простого и будем двигаться к сложному.

Глобальное совместное использование переменной

Для начала самый простой вариант — глобально «поделиться» переменной с каждым представлением в вашем приложении, как в примере 4.22.

Пример 4.22. Глобальное совместное использование переменной

```
// Некий сервис-провайдер
public function boot()
{
    ...
    view()->share('recentPosts', Post::recent());
}
```

Если вы хотите использовать `view()->share()`, лучшим решением будет `boot()` сервис-провайдера, чтобы привязка выполнялась при каждой загрузке страницы. Вы можете создать собственный `ViewComposerServiceProvider` (подробнее о сервис-провайдерах см. в главе 11), а пока просто поместите `view()->share()` в `App\Providers\AppServiceProvider` в метод `boot()`.

Использование `view()->share()` делает переменную доступной для каждого представления во всем приложении, но это может быть излишним.

Компоновщики с замыканиями с областью видимости масштаба представления

Следующий вариант — использовать компоновщик представлений на основе замыканий для совместного применения переменных с одним представлением, как в примере 4.23.

Пример 4.23. Создание основанного на замыкании компоновщика представления

```
view()->composer('partials.sidebar', function ($view) {
    $view->with('recentPosts', Post::recent());
});
```

Как видите, в первом параметре (`partials.sidebar`) мы определили имя представления, с которым хотим поделиться, а затем передали замыкание второму параметру. В замыкании мы использовали `$view->with()` для совместного применения переменной, но только для конкретного представления.

КОМПОНОВЩИКИ ПРЕДСТАВЛЕНИЙ ДЛЯ НЕСКОЛЬКИХ ПРЕДСТАВЛЕНИЙ

Везде, где компоновщик представлений привязывается к определенному представлению (как в примере 4.23, где компоновщик привязывается к `partials.sidebar`), вы можете передавать массив имен представлений, а не привязываться к нескольким представлениям.

Можно также использовать звездочку в пути представления, например, `partials.*` или `tasks.*`:

```
view()->composer(
    ['partials.header', 'partials.footer'],
    function () {
        $view->with('recentPosts', Post::recent());
    }
);

view()->composer('partials.*', function () {
    $view->with('recentPosts', Post::recent());
});
```

Компоновщики представлений с областью видимости масштаба представления и классами

Наконец, самый гибкий, но и самый сложный вариант — создать отдельный класс для вашего компоновщика представления.

Создадим класс компоновщика представления. Формально определенного места нет, но в документах рекомендуется `App\Http\ViewComposers`. Итак, создадим `App\Http\ViewComposers\RecentPostsComposer`, как в примере 4.24.

Пример 4.24. Компоновщик представления

```
<?php

namespace App\Http\ViewComposers;

use App\Post;
use Illuminate\Contracts\View\View;

class RecentPostsComposer
{
    public function compose(View $view)
    {
        $view->with('recentPosts', Post::recent());
    }
}
```

Когда вызывается этот компоновщик, он запускает метод `compose()`, в котором мы связываем переменную `recentPosts` с результатом запуска метода `recent()` модели `Post`.

Как и другие методы совместного использования переменных, этот компоновщик представления должен быть где-то привязан. Скорее всего, вы создадите пользовательский `ViewComposerServiceProvider`, но сейчас, как видно из примера 4.25, мы просто поместим его в метод `boot()` `App\Providers\AppServiceProvider`.

Пример 4.25. Регистрация компоновщика представления в `AppServiceProvider`

```
public function boot()
{
    view()->composer(
        'partials.sidebar',
        \App\Http\ViewComposers\RecentPostsComposer::class
    );
}
```

Обратите внимание, что эта привязка отличается от компоновщика представлений на основе замыканий лишь тем, что вместо замыкания мы передаем имя класса компоновщика представлений. Теперь каждый раз, когда **Blade** отображает представление `partials.sidebar`, он автоматически запускает наш провайдер и передает представлению переменную `RecentPosts` с результатами метода `recent()` нашей модели `Post`.

Внедрение сервиса Blade

Существует три основных типа данных, которые возможно внедрить в представление: коллекции данных для итерации; отдельные объекты, отображаемые на странице; сервисы, которые генерируют данные или представления.

В случае сервиса шаблон, вероятно, будет выглядеть как в примере 4.26, где мы добавляем экземпляр нашего аналитического сервиса в определение маршрута, указывая тип в сигнатуре его метода, а затем передаем в представление.

Пример 4.26. Внедрение сервисов в представление через конструктор определения маршрута

```
Route::get('backend/sales', function (AnalyticsService $analytics) {
    return view('backend.sales-graphs')
        ->with('analytics', $analytics);
});
```

Как и в случае с компоновщиками представлений, внедрение сервиса Blade предлагает удобную сокращенную форму для уменьшения дублирования в ваших определениях маршрутов. Обычно содержимое представления с использованием нашего аналитического сервиса может выглядеть как пример 4.27.

Пример 4.27. Использование внедренного навигационного сервиса в представлении

```
<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

Внедрение сервиса Blade позволяет легко добавить экземпляр класса из контейнера напрямую из представления, как в примере 4.28.

Пример 4.28. Внедрение сервиса напрямую в представление

```
@inject('analytics', 'App\Services\Analytics')

<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

Директива `@inject` фактически сделала доступной переменную `$analytics`, что мы будем использовать позже в нашем представлении.

Первый параметр `@inject` — это имя внедряемой переменной, а второй параметр — класс или интерфейс, откуда вы хотите внедрить экземпляр. Это разрешается так же, как если бы вы указали тип зависимости в конструкторе где-нибудь в Laravel; если вы не знакомы с тем, как это работает, обратитесь к главе 11.

Подобно компоновщикам представлений, добавление сервисов Blade позволяет легко сделать конкретные данные или функциональные возможности доступными для каждого экземпляра представления без необходимости каждый раз внедрять их через определение маршрута.

Пользовательские директивы Blade

Весь встроенный синтаксис Blade, который мы рассматривали до сих пор — `@if`, `@unless` и т. п., — называется *директивами*. Это отображение между шаблоном (например, `@if ($condition)`) и выводом PHP (скажем, `<?php if ($condition): ?>`).

Директивы бывают не только встроенными — можно создать и собственную. Вы можете подумать, что директивы хороши для создания кратких форм для больших участков кода — например, с использованием `@button('buttonName')` и расширением его до большого набора кнопок HTML. Это *неплохая* идея, но, возможно, лучше включить составляющую представления.

Пользовательские директивы особенно полезны, когда с их помощью упрощается некоторая форма повторяющейся логики. Скажем, мы устали от необходимости оборачивать наш код с помощью `@if (auth()->guest())` (проверить, вошел ли пользователь в систему), и нам нужна специальная директива `@ifGuest`. Как и в случае с компоновщиками представлений, может быть, стоит зарегистрировать директиву в пользовательском сервис-провайдере, но сейчас просто поместим ее в метод `boot()` `App\Providers\AppServiceProvider`. В примере 4.29 показано, как будет выглядеть привязка.

Пример 4.29. Привязка пользовательской директивы Blade в сервис-провайдере

```
public function boot()
{
    Blade::directive('ifGuest', function () {
        return "<?php if (auth()->guest()): ?>";
    });
}
```

Теперь мы зарегистрировали пользовательскую директиву `@ifGuest`, которая будет заменена РНР-кодом `<?php if (auth()->guest()): ?>`.

Это может показаться странным. Вы пишете *строку*, которая будет возвращена и затем выполнена как РНР. Но это означает, что теперь вы можете взять сложные, некрасивые, неясные или повторяющиеся фрагменты вашего стереотипного кода РНР и скрыть их за ясным, простым и выразительным синтаксисом.



Кэширование результатов пользовательской директивы

Может возникнуть соблазн использовать некий алгоритм, чтобы ускорить выполнение вашей пользовательской директивы, сделав операцию в привязке, а затем внедрив результат в возвращаемую строку:

```
Blade::directive('ifGuest', function () {
    // Антишаблон! Не копируйте
    $ifGuest = auth()->guest();
    return "<?php if ({${ifGuest}}): ?>";
});
```

Проблема в том, что предполагается, что эта директива будет воссоздаваться при каждой загрузке страницы. Однако Blade кэширует агрессивно, поэтому с таким подходом вы окажетесь в плохом положении.

Параметры пользовательских директив Blade

Что, если вы хотите принять параметры в вашем пользовательском алгоритме? Посмотрите на пример 4.30.

Пример 4.30. Создание директивы Blade с параметрами

```
// Связывание
Blade::directive('newlinesToBr', function ($expression) {
    return "<?php echo nl2br({$expression}); ?>";
});

// В использовании
<p>@newlinesToBr($message->body)</p>
```

Параметр `$expression`, полученный замыканием, представляет все, что находится в скобках. Как видите, затем мы генерируем правильный фрагмент кода PHP и возвращаем его.



Область видимости параметра `$expression` до Laravel 5.3

До Laravel 5.3 параметр `$expression` также включал сами скобки. Таким образом, в примере 4.30 `$expression` (то есть `$message->body` в версии 5.3 и более поздних) вместо этого было бы `($message->body)` и пришлось бы написать `<?php echo nl2br({$expression}); ?>`.

Если вы в который раз пишете один и тот же условный алгоритм, следует рассмотреть использование директивы Blade.

Пример: применение пользовательских директив Blade для многоклиентских приложений

Представим, что мы создаем приложение для поддержки работы с несколькими клиентами — пользователи могут посещать сайты `www.myapp.com`, `client1.myapp.com`, `client2.myapp.com` или другие.

Предположим, мы написали класс для инкапсуляции некоторой логики многоклиентской работы и назвали его `Context`. Он будет собирать информацию и логику о контексте текущего посещения, например, кто прошел проверку пользователем и посещает ли он публичный сайт/клиентский поддомен.

Мы, вероятно, будем часто разрешать этот класс `Context` в наших представлениях и выполнять его условия так, как в примере 4.31. `app('context')` — это сокращенная форма для получения экземпляра класса из контейнера (узнаем больше в главе 11).

Пример 4.31. Условные конструкции для контекста без пользовательской директивы Blade

```
@if (app('context')->isPublic())
    &copy; Copyright MyApp LLC
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

А если бы мы могли упростить `@if (app('context')->isPublic())` до `@ifPublic?` Давайте попробуем. Посмотрите на пример 4.32.

Пример 4.32. Условная конструкция для контекста с пользовательской директивой Blade

```
// Связывание
Blade::directive('ifPublic', function () {
    return "<?php if (app('context')->isPublic()): ?>";
});

// В использовании
@ifPublic
    &copy; Copyright MyApp LLC
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

Поскольку это переводится в простой оператор `if`, мы все еще можем полагаться на родные условные выражения `@else` и `@endif`. Но при желании могли бы также создать собственную директиву `@elseifClient`, или отдельную `@ifClient`, или действительно все, что мы хотим.

Упрощенные пользовательские директивы для операторов `if`

5.5 Хотя пользовательские директивы Blade — мощное средство, для них чаще используются операторы `if`. Таким образом, существует более простой способ создания пользовательских директив `if`: `Blade::if()`. В примере 4.33 показано, как мы могли бы перестроить пример 4.32 с применением метода `Blade::if()`.

Пример 4.33. Определение пользовательской директивы `if`

```
// Привязка
Blade::if('ifPublic', function () {
    return (app('context')->isPublic());
});
```

Вы будете использовать директивы точно так же, но определить их немного проще. Вместо того чтобы вручную набирать скобки РНР, вы можете просто написать замыкание, которое возвращает логическое значение.

Тестирование

Самый распространенный метод испытания представлений — тестирование приложений, то есть вы фактически вызываете маршрут, который отображает представления, и убеждаетесь, что они имеют определенное содержимое (пример 4.34). Можно нажимать кнопки или отправлять формы и убедиться, что вы перенаправлены на определенную страницу или видите определенную ошибку. Подробнее о тестировании поговорим в главе 12.

Пример 4.34. Проверка того, что представление отображает определенный контент

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();
    $event2 = factory(Event::class)->create();

    $this->get('events')
        ->assertSee($event1->title)
        ->assertSee($event2->title);
}
```

Можно проверить, что конкретному представлению был передан определенный набор данных. При соответствии вашим целям тестирования это менее рискованно, чем проверка конкретного текста на странице. Пример 4.35 демонстрирует такой подход.

Пример 4.35. Тестирование того, что представлению было передано определенное содержимое

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();
    $event2 = factory(Event::class)->create();

    $response = $this->get('events');

    $response->assertViewHas('events', Event::all());
    $response->assertViewHasAll([
        'events' => Event::all(),
        'title' => 'Events Page',
    ]);
    $response->assertViewMissing('dogs');
}
```



Названия методов тестирования в Laravel до версии 5.4

В проектах, работающих с версиями Laravel до 5.4, `get()` и `assertSee()` должны быть заменены на `visit()` и `see()`.

В версии 5.3 появилась возможность передавать замыкание в `assertViewHas()`. Так мы можем настроить способ проверки более сложных структур данных. Пример 4.36 показывает, как это можно использовать.

Пример 4.36. Передача замыкания в `assertViewHas()`

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();

    $response = $this->get("events/{ $event1->id }");

    $response->assertViewHas('event', function ($event) use ($event1) {
        return $event->id === $event1->id;
    });
}
```

Резюме

Blade — это движок шаблонизации Laravel. Он фокусируется на четком, кратком и выразительном синтаксисе с мощным наследованием и расширяемостью. Его скобки «безопасного Echo» — `{{ и }}`, а скобки незащищенного — `{!! и !!}`. Есть ряд пользовательских тегов, называемых директивами, которые начинаются с `@` (например, `@if` и `@unless`).

Вы можете определить родительский шаблон и оставить в нем пустые места для содержимого, добавив `@yield` и `@section/@show`. После этого с помощью директивы `@extends('parent.view')` можно «научить» дочерние представления расширять этот шаблон и определить их разделы, добавив `@section/@endsection`. Для ссылки на содержимое тех или иных блоков в родительском представлении используется директива `@parent`.

Компоновщики представлений позволяют легко задать, что при каждой загрузке определенного представления или подпредставления ему должна быть доступна конкретная информация. Внедрение сервиса дает возможность самому представлению запрашивать данные прямо из контейнера приложения.

5

Базы данных и Eloquent

Laravel предоставляет целый набор инструментов для взаимодействия с базами данных вашего приложения, но наиболее заметные из них — Eloquent и ActiveRecord ORM Laravel.

Eloquent — один из самых популярных и полезных компонентов фреймворка. Это хороший пример отличия Laravel от большинства сред PHP в мире мощных, но сложных DataMapper ORM, Eloquent отличается простотой. Для каждой таблицы существует один класс, который отвечает за извлечение, представление и сохранение данных в этой таблице.

Независимо от того, решите вы использовать Eloquent или нет, вы все равно получите массу преимуществ от работы с другими инструментами баз данных, которые предоставляет Laravel. Итак, сначала рассмотрим основы функциональности: миграции, сидеры и генератор запросов.

Затем разберем Eloquent: определение ваших моделей; вставку, обновление и удаление; настройку ваших ответов с помощью методов доступа, модифицирующих методов и атрибутов; отношения. Так много всего, что легко растеряться, но шаг за шагом мы все рассмотрим.

Конфигурация

Прежде чем мы начнем изучать использование инструментов баз данных Laravel, сделаем паузу и рассмотрим, как настроить учетные данные и соединения вашей базы данных.

Конфигурация доступа к базе данных находится в файлах `config/database.php` и `.env`. Как и во многих других конфигурациях в Laravel, можно определить несколько «соединений», а затем решить, какой код будет применяться по умолчанию.

Подключения базы данных

По умолчанию для каждого из драйверов задано одно соединение, как вы можете видеть в примере 5.1.

Пример 5.1. Список подключений к базе данных по умолчанию

```
'connections' => [  
  'sqlite' => [  
    'driver' => 'sqlite',  
    'database' => env('DB_DATABASE', database_path('database.sqlite')),  
    'prefix' => '',  
  ],  
  'mysql' => [  
    'driver' => 'mysql',  
    'host' => env('DB_HOST', '127.0.0.1'),  
    'port' => env('DB_PORT', '3306'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'unix_socket' => env('DB_SOCKET', ''),  
    'charset' => 'utf8',  
    'collation' => 'utf8_unicode_ci',  
    'prefix' => '',  
    'strict' => false,  
    'engine' => null,  
  ],  
  'pgsql' => [  
    'driver' => 'pgsql',  
    'host' => env('DB_HOST', '127.0.0.1'),  
    'port' => env('DB_PORT', '5432'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
    'schema' => 'public',  
    'sslmode' => 'prefer',  
  ],  
  'sqlsrv' => [  
    'driver' => 'sqlsrv',  
    'host' => env('DB_HOST', 'localhost'),  
    'port' => env('DB_PORT', '1433'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
  ],  
]
```

Ничто не мешает вам удалять/изменять эти именованные соединения или создавать собственные, устанавливая в них драйверы (MySQL, Postgres и т. д.). Таким образом, по умолчанию для каждого драйвера утверждено одно соединение, но это не ограничение. У вас может быть пять разных соединений, все с драйвером `mysql`, если хотите.

Можно определять свойства для подключения и настройки каждого типа соединения.

Есть причины для идеи нескольких драйверов. Начнем с того, что раздел «соединения» — это простой шаблон, который позволяет легко запускать приложения, использующие любой из поддерживаемых типов подключений с базой данных. Во многих приложениях вы можете выбрать соединение с базой данных, которое будете применять, заполнить его информацией и даже удалить остальные, если хотите. Обычно я просто держу их вместе, вдруг понадобится использовать.

Иногда нужно несколько подключений в одном приложении. Например, вы можете использовать разные соединения с базой данных для двух разных типов данных или читать из одного соединения, а записывать через другое. В этом поможет поддержка нескольких соединений.

Другие параметры конфигурации базы данных

В разделе конфигурации файла `config/database.php` есть довольно много иных параметров конфигурации. Вы можете настроить доступ Redis и имя таблицы, используемой для миграций, определить подключение по умолчанию и указать, возвращают ли не-Eloquent-вызовы `stdClass` или экземпляры массива.

С любым сервисом в Laravel, который позволяет соединения из нескольких источников, сессии могут поддерживаться базой данных или хранилищем файлов. Кэш может использовать Redis или Memcached, базы данных могут применять MySQL или PostgreSQL. Вы можете определить несколько соединений, а также выбрать конкретное соединение по умолчанию, то есть то, которое будет использоваться каждый раз, когда вы явно не запрашиваете конкретное соединение. Так можно запросить конкретное соединение, если хотите:

```
$users = DB::connection('secondary')->select('select * from users');
```

Миграции

Современные фреймворки вроде Laravel позволяют легко определять структуру вашей базы данных с помощью миграций на основе кода. Каждая новая таблица, столбец, индекс и ключ могут определяться в коде, а любая новая среда — переноситься из пустой базы данных в идеальную схему вашего приложения за считанные секунды.

Определение миграций

Миграция — это отдельный файл, определяющий модификации, которые необходимо выполнить при запуске этой миграции *наверх* и *вниз* (не обязательно).

МЕТОДЫ UP() И DOWN() В МИГРАЦИЯХ

Миграции всегда выполняются по порядку по дате. Каждый файл миграции называется примерно так: `2018_10_12_000000_create_users_table.php`. Когда выполняется миграция новой системы, она захватывает каждую миграцию, начиная с самой ранней даты, и запускает ее метод `up()` — в этот момент вы переносите ее «наверх». Но можно «откатить» ваш последний набор миграций: захватив каждую из них и запустив метод `down()`, который должен отменить все изменения, сделанные при сдвиге «вверх».

Таким образом, метод `up()` миграции должен «выполнять» перенос, а `down()` — «отменять» его.

Пример 5.2 показывает, как выглядит миграция по умолчанию «создать таблицу пользователей», которая поставляется с Laravel.

Пример 5.2. Миграция Laravel по умолчанию «создать таблицу пользователей»

```
<?php
```

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

class CreateUsersTable extends Migration
{
    /**
     * Запустить миграции.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Откатить миграции.
     */
}
```

```

    * @return void
    */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}

```



Подтверждение адреса электронной почты

 Столбец `email_verified_at` есть только в приложениях Laravel 5.7 и более поздних версий. Он хранит отметку времени, указывающую, когда пользователь подтвердил свой адрес электронной почты.

Как видите, у нас есть методы `up()` и `down()`. Первый говорит миграции создать новую таблицу с именем `users` с несколькими полями, а второй — удалить ее.

Создание миграции

Как вы увидите в главе 8, Laravel предоставляет ряд инструментов командной строки для взаимодействия с вашим приложением и создания шаблонных файлов. Одна из этих команд позволяет создать файл миграции. Запустить ее можно командой `php artisan make:migration` с единственным параметром — именем миграции. Например, чтобы создать рассмотренную таблицу, нужно ввести `php artisan make:migration create_users_table`.

Есть два флага, которые вы можете опционально передать этой команде. `--create=table_name` предварительно заполняет миграцию кодом, предназначенным для создания таблицы с именем `table_name`, а `--table=table_name` — для изменений в существующей таблице. Вот несколько примеров:

```

php artisan make:migration create_users_table
php artisan make:migration add_votes_to_users_table --table=users
php artisan make:migration create_users_table --create=users

```

Создание таблиц

Мы уже видели в миграции по умолчанию `create_users_table`, что наши миграции зависят от фасада `Schema` и его методов. Все действия здесь будут основываться на методах `Schema`.

Чтобы создать новую таблицу в процессе миграции, используйте метод `create()`, в котором первый параметр — это имя таблицы, а второй — замыкание, определяющее ее столбцы:

```

Schema::create('users', function (Blueprint $table) {
    // Здесь создаем столбцы
});

```


Создание столбцов

Для новых столбцов в таблице, будь то при вызове создания таблицы или при вызове ее изменения, используйте экземпляр **Blueprint**, переданный в ваше замыкание:

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

Посмотрим на различные методы, доступные в экземплярах **Blueprint** для создания столбцов. Я опишу, как они работают в MySQL. Если вы используете другую базу данных, Laravel просто применит ее эквивалент.

Ниже приведены простые полевые методы **Blueprint**.

- ❑ `integer(colName)`, `tinyInteger(colName)`, `smallInteger(colName)`, `mediumInteger(colName)`, `bigInteger(colName)`. Добавляет столбец типа **INTEGER** или один из его многочисленных вариантов.
- ❑ `string(colName, length)`. Добавляет столбец типа **VARCHAR** и — необязательно — длину.
- ❑ `binary(colName)`. Добавляет столбец типа **BLOB**.
- ❑ `boolean(colName)`. Добавляет столбец типа **BOOLEAN** (**TINYINT(1)** в MySQL).
- ❑ `char(colName, length)`. Добавляет столбец **CHAR** с необязательной длиной.
- ❑ `datetime(colName)`. Добавляет столбец **DATETIME**.
- ❑ `decimal(colName, precision, scale)`. Добавляет столбец **DECIMAL** с точностью и масштабом — например, `decimal('amount', 5, 2)` задает точность 5 и масштаб 2.
- ❑ `double(colName, total digits, digits after decimal)`. Добавляет столбец **DOUBLE** — например, `double('tolerance', 12, 8)` задает длину 12 цифр, причем восемь из этих цифр справа от десятичного знака: **7204.05691739**.
- ❑ `enum(colName, [choiceOne, choiceTwo])`. Добавляет столбец **ENUM** с предоставленными вариантами.
- ❑ `float(colName, precision, scale)`. Добавляет столбец **FLOAT** (то же самое, что **double** в MySQL).
- ❑ `json(colName)` и `jsonb(colName)`. Добавляет столбец **JSON** или **JSONB** (или столбец **TEXT** в Laravel 5.1).
- ❑ `text(colName)`, `mediumText(colName)`, `longText(colName)`. Добавляет столбец **TEXT** (или его вариации с различными размерами).
- ❑ `time(colName)`. Добавляет столбец **TIME**.
- ❑ `timestamp(colName)`. Добавляет столбец **TIMESTAMP**.
- ❑ `uuid(colName)`. Добавляет столбец **UUID** (**CHAR(36)** в MySQL).

А это специальные (соединенные) методы Blueprint.

- ❑ `increments(colName)` и `bigIncrements(colName)`. Добавляет идентификатор первичного ключа беззнакового инкрементного `INTEGER` или `BIG INTEGER`.
- ❑ `timestamps()` и `nullableTimestamps()`. Добавляет столбцы меток времени `created_at` и `updated_at`.
- ❑ `rememberToken()`. Добавляет столбец `remember_token` (`VARCHAR(100)`) для пользовательских токенов «запомнить меня».
- ❑ `softDeletes()`. Добавляет метку времени `deleted_at` для использования с мягкими удалениями.
- ❑ `morphs(colName)`. Для указанной `colName` добавляет целое число `colName_id` и строку `colName_type` (например, `morphs(tag)` добавляет целое число `tag_id` и строку `tag_type`); предназначен для использования в полиморфных связях.

Гибкое создание дополнительных свойств

Большинство свойств определения поля — скажем, его длина — задается как второй параметр метода создания поля (см. предыдущий раздел). Но те, которые мы зададим, вызывают больше методов после создания столбца. Например, поле `email` при нулевом значении будет помещено (в MySQL) сразу после `last_name`:

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

Следующие методы используются для установки дополнительных свойств поля.

- ❑ `nullable()`. Позволяет вставлять в этот столбец значения `NULL`.
- ❑ `default('default content')`. Указывает содержимое по умолчанию для этого столбца, если значение не указано.
- ❑ `unsigned()`. Помечает целочисленные столбцы как беззнаковые (не отрицательные или положительные, а просто целые).
- ❑ `first()` (*только в MySQL*). Помещает столбец первым в порядке столбцов.
- ❑ `after(colName)` (*только в MySQL*). Размещает столбец после другого столбца в порядке столбцов.
- ❑ `unique()`. Добавляет индекс `UNIQUE`.
- ❑ `primary()`. Добавляет индекс первичного ключа.
- ❑ `index()`. Добавляет базовый индекс.

Обратите внимание, что `unique()`, `primary()` и `index()` также могут быть вне контекста текущего построения столбцов, что мы рассмотрим позже.

Уничтожение таблиц

Если вы хотите удалить таблицу, воспользуйтесь методом `dropIfExists()`, который принимает один параметр — имя таблицы:

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

Изменение столбцов

Чтобы изменить столбец, напишите код (как для создания нового столбца), а после добавьте вызов метода `change()`.



Требуемая зависимость перед изменением столбцов

Прежде чем изменять какие-либо столбцы (или удалять в SQLite), необходимо запустить `composer require doctrine/dbal`.

Итак, если у нас есть строковый столбец с именем `name` длиной 255, а мы хотим изменить ее на 100, пишем:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 100)->change();
});
```

То же самое верно, если мы хотим добавить какие-либо свойства, которые не определены в имени метода. Чтобы сделать возможным нулевые значения в полях, мы напишем:

```
Schema::table('contacts', function (Blueprint $table) {
    $table->string('deleted_at')->nullable()->change();
});
```

Вот так можно переименовать столбец:

```
Schema::table('contacts', function (Blueprint $table) {
    $table->renameColumn('promoted', 'is_promoted');
});
```

А так — удалить столбец:

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



Изменение нескольких столбцов одновременно в SQLite

Если вы попытаетесь удалить или изменить несколько столбцов в одном замыкании миграции и воспользуетесь SQLite, то столкнетесь с ошибками.

В главе 12 я рекомендую использовать SQLite для тестовой базы данных, поэтому даже с более традиционной базой данных стоит рассмотреть это ограничение для целей тестирования.

Тем не менее вам не нужно создавать новую миграцию каждый раз. Вместо этого просто сделайте несколько вызовов `Schema::table()` в методе `up()` вашей миграции:

```
public function up()
{
    Schema::table('contacts', function (Blueprint $table)
    {
        $table->dropColumn('is_promoted');
    });
    Schema::table('contacts', function (Blueprint $table)
    {
        $table->dropColumn('alternate_email');
    });
}
```

Индексы и внешние ключи

Мы рассмотрели создание, изменение и удаление столбцов. Перейдем к индексации и установлению связей.

Ваши базы данных могут работать, даже если вы никогда их не используете, но они очень важны для оптимизации производительности и некоторых средств контроля целостности данных в отношении связанных таблиц. Я бы порекомендовал почитать о них, при острой необходимости можно пропустить этот раздел.

Добавление индексов. Посмотрите в примере 5.3, как добавить индексы в ваш столбец.

Пример 5.3. Добавление индексов столбцов в миграциях

```
// После того, как столбцы созданы...
$table->primary('primary_id'); // Первичный ключ;
                                // не нужно, если используется increments()
$table->primary(['first_name', 'last_name']); // Составные ключи
$table->unique('email'); // Уникальный индекс
$table->unique('email', 'optional_custom_index_name'); // Уникальный индекс
$table->index('amount'); // Базовый индекс
$table->index('amount', 'optional_custom_index_name'); // Базовый индекс
```

В первом примере, `primary()`, нет необходимости, если вы используете методы `increments()` или `bigIncrements()` для создания своего индекса. Индекс первичного ключа добавится автоматически.

Удаление индексов. Мы можем удалить индексы, как показано в примере 5.4.

Пример 5.4. Удаление индексов столбцов в миграциях

```
$table->dropPrimary('contacts_id_primary');
$table->dropUnique('contacts_email_unique');
$table->dropIndex('optional_custom_index_name');

// Если вы передадите массив имен столбцов в dropIndex, он будет делать
// предположения об именах индексов для вас на основе правил генерации
$table->dropIndex(['email', 'amount']);
```

Добавление и удаление внешних ключей. Синтаксис Laravel для добавления внешнего ключа, определяющего, что конкретный столбец ссылается на столбец в другой таблице, прост и понятен:

```
$table->foreign('user_id')->references('id')->on('users');
```

Здесь мы добавляем `foreign` индекс по столбцу `user_id`, показывая, что он ссылается на столбец `id` в таблице `users`. Легко.

Если мы хотим указать ограничения внешнего ключа, можно использовать `onDelete()` и `onUpdate()`. Например:

```
$table->foreign('user_id')
    ->references('id')
    ->on('users')
    ->onDelete('cascade');
```

Чтобы отбросить внешний ключ, можно удалить его, сославшись на его индексное имя (автоматически генерируется путем объединения имен столбцов и таблиц, на которые он ссылается):

```
$table->dropForeign('contacts_user_id_foreign');
```

Либо передав ему массив полей, на которые он ссылается в локальной таблице:

```
$table->dropForeign(['user_id']);
```

Запуск миграций

Как их запустить после определения своей миграции? Для этого есть команда Artisan:

```
php artisan migrate
```

Она запускает все невыполненные миграции (методом `up()` для каждой). Laravel отслеживает, какие миграции вы выполнили, а какие — нет. При вводе Artisan всегда проверяет выполнение всех доступных миграций и запускает оставшиеся.

В этом пространстве имен есть несколько опций для работы. Можно запустить ваши миграции и сидеры (о чем расскажем далее):

```
php artisan migrate --seed
```

Вы также можете выполнить любую из следующих команд.

- ❑ **migrate:install.** Создает таблицу базы данных, которая отслеживает выполнение миграций. Команда запускается автоматически при миграции, поэтому можете игнорировать ее.
- ❑ **migrate:reset.** Откатывает каждую миграцию базы данных, которую вы запускали на этом экземпляре.
- ❑ **migrate:refresh.** Откатывает каждую миграцию базы данных, которую вы запускали на этом экземпляре, а затем запускает каждую доступную миграцию. Это то же самое, что выполнить **migrate:reset**, а затем **migrate**.
- ❑ **migrate:fresh.** Удаляет все ваши таблицы и запускает каждую миграцию снова. Это то же самое, что **refresh**, но без миграции «вниз». Команда просто удаляет таблицы, а затем снова запускает миграцию «вверх».
- ❑ **migrate:rollback.** Откатывает *только* последние миграции при последней миграции или, с добавленным параметром **--step=n**, откатывает указанное вами количество миграций.
- ❑ **migrate:status.** Показывает таблицу со всеми миграциями, рядом с каждой стоит Y или N, обозначающие выполнение миграций в этой среде.



Миграция с Homestead/Vagrant

Если при миграции на локальном компьютере ваш файл `.env` указывает на базу данных в окне Vagrant, то миграции завершатся неудачно. Нужно зайти в свой Vagrant-бокс по `ssh` и запустить миграцию. То же самое для заполнения и любых других команд Artisan, которые что-то делают с базой данных или читают из нее.

Наполнение базы данными

Наполнение (seeding) баз данных с помощью Laravel — простая функция, поэтому очень часто используется в процессе разработки, как это было в предыдущих платформах PHP. Существует папка **database/seeds**, которая поставляется с классом **DatabaseSeeder**. У него есть метод `run()`, который вызывается с сидером (seeder).

Два основных способа запуска сидеров: вместе с миграцией или отдельно.

Чтобы запустить сидер вместе с миграцией, добавьте **--seed** к любому вызову миграции:

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

Чтобы запустить сидер независимо:

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

Это вызовет метод `run()` объекта `DatabaseSeeder` по умолчанию или класс сидера, заданный параметром `--class`.

Создание сидера

Сидер создается командой `Artisan make:seeder`:

```
php artisan make:seeder ContactsTableSeeder
```

Теперь у вас появится класс `ContactsTableSeeder`, расположенный в каталоге `database/seeds`. Прежде чем редактировать, добавим его в `DatabaseSeeder`, как показано в примере 5.5. Он будет запускаться с нашими сидерами.

Пример 5.5. Вызов пользовательского сидера из `DatabaseSeeder.php`

```
// database/seeds/DatabaseSeeder.php
...
    public function run()
    {
        $this->call(ContactsTableSeeder::class);
    }
```

Теперь отредактируем сам сидер. Самое простое, что мы можем сделать, — это вручную вставить запись, используя фасад `DB`, как показано в примере 5.6.

Пример 5.6. Вставка записей базы данных в пользовательский сидер

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class ContactsTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('contacts')->insert([
            'name' => 'Lupita Smith',
            'email' => 'lupita@gmail.com',
        ]);
    }
}
```

Это даст нам единственную запись, что хорошо для начала. Но ради действительно функционального заполнения вы, вероятно, захотите запустить в цикле какой-нибудь генератор случайных чисел и много раз выполнить этот `insert()`. Laravel предоставляет такую возможность.

Фабрики моделей

Фабрики моделей определяют один/несколько шаблонов создания фейковых записей для таблиц вашей базы данных. По умолчанию каждая фабрика названа по классу Eloquent, но можно и по имени таблицы, если вы не собираетесь работать с Eloquent. В примере 5.7 показаны оба варианта настройки фабрики.

Пример 5.7. Определение фабрик моделей с помощью ключей имен классов Eloquent и таблиц

```
$factory->define(User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
    ];
});

$factory->define('users', function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
    ];
});
```

Теоретически вы можете назвать эти фабрики как угодно, но лучше всего — по имени вашего класса Eloquent.

Создание фабрики моделей

5.5 Модельные фабрики расположены в `database/factories`. В Laravel 5.5 и более поздних версиях каждая фабрика обычно определяется в собственном классе с ключом (именем) и замыканием, устанавливающим, как создать новый экземпляр конкретного класса. Метод `$factory->define()` принимает название фабрики в качестве первого параметра и замыкание, которое выполняется при каждой генерации, как второй.



Файл фабрики моделей в Laravel 5.4 и ранее

В Laravel до 5.5 нет отдельных классов для каждой фабрики, поэтому они должны определяться в файле `database/factories/ModelFactory.php`.

Чтобы сгенерировать новый класс фабрики, используйте команду `Artisan make:factory`. Как и при именовании ключей фабрики, классы фабрик чаще всего называют как модели Eloquent, для генерации экземпляров которых они предназначены:

```
php artisan make:factory ContactFactory
```

Это создаст новый файл в каталоге `database/factories` с именем `ContactFactory.php`. Самая простая фабрика, которую мы можем определить для контакта, может выглядеть так, как показано в примере 5.8.

Пример 5.8. Самое простое определение фабрики

```
$factory->define(Contact::class, function (Faker\Generator $faker) {  
    return [  
        'name' => 'Lupita Smith',  
        'email' => 'lupita@gmail.com',  
    ];  
});
```

Теперь мы можем использовать глобальный хелпер `factory()` для создания экземпляра `Contact` в нашем заполнении и тестировании:

```
// Создать один  
$contact = factory(Contact::class)->create();  
  
// Создать множество  
factory(Contact::class, 20)->create();
```

Если бы мы применили эту фабрику для создания 20 контактов, они имели бы одинаковую информацию. Это не очень полезно.

Мы получим большую выгоду от фабрик моделей, если воспользуемся интерфейсом `Faker`, переданным в замыкание. Интерфейс `Faker` позволяет легко создавать случайные структурированные поддельные данные. Предыдущий пример теперь превращается в пример 5.9.

Пример 5.9. Простая фабрика, модифицированная для использования `Faker`

```
$factory->define(Contact::class, function (Faker\Generator $faker) {  
    return [  
        'name' => $faker->name,  
        'email' => $faker->email,  
    ];  
});
```

Теперь каждый раз при создании фальшивого контакта с помощью этой фабрики моделей все наши свойства будут генерироваться случайным образом.



Гарантия уникальности случайно сгенерированных данных

Если вы хотите гарантировать, что случайно сгенерированные значения любой данной записи уникальны, по сравнению с другими случайно сгенерированными значениями во время этого процесса PHP, можно использовать метод `Faker unique()`:

```
return ['email' => $faker->unique()->email];
```

Использование фабрики моделей

Существует два основных контекста, в которых мы будем использовать фабрики моделей: тестирование (см. главу 12) и заполнение. Напишем сидер с применением фабрики моделей (пример 5.10).

Пример 5.10. Использование фабрик моделей

```
$post = factory(Post::class)->create([
    'title' => 'My greatest post ever',
]);

// Фабрика профессионального уровня, но вам не стоит опасаться!
factory(User::class, 20)->create()->each(function ($u) use ($post) {
    $post->comments()->save(factory(Comment::class)->make([
        'user_id' => $u->id,
    ]))
});
```

Чтобы создать объект, мы обращаемся к глобальному хелперу `factory()` и передаем ему имя фабрики, которое совпадает с названием класса Eloquent, экземпляр которого мы генерируем. Он возвращает фабрику, и тогда мы можем запустить один из двух методов: `make()` или `create()`.

Оба метода генерируют экземпляр этой указанной модели, используя определение в файле фабрики. Разница в том, что `make()` создает экземпляр, но (пока) не сохраняет его в базе данных, тогда как `create()` сохраняет сразу же. Вы можете увидеть оба варианта в примере 5.10.

Второй метод станет понятнее, когда мы рассмотрим отношения в Eloquent позже в этой главе.

Переопределение свойств при вызове фабрики моделей. При передаче массива в `make()` или `create()` вы можете переопределить некоторые ключи фабрики (см. пример 5.10), чтобы установить `user_id` в комментарии и вручную задать заголовки нашего сообщения.

Создание более одного экземпляра с фабрикой моделей. Если вы передаете число в качестве второго параметра хелперу `factory()`, значит, вы создаете больше одного экземпляра. Вместо того чтобы возвращать один экземпляр, хелпер возвращает целую коллекцию. Следовательно, вы можете рассматривать результат как массив, связать каждый из его экземпляров с другой сущностью или использовать другие методы сущности для каждого экземпляра (см. `each()` в примере 5.10), чтобы добавить комментарий по каждому вновь созданному пользователю.

Фабрики моделей профессионального уровня. Теперь, когда мы ознакомились с наиболее распространенными использованием и расстановкой фабрик моделей, рассмотрим некоторые из более сложных способов их применения.

Присоединение отношений при определении модельных фабрик. Иногда вам нужно создать связанный элемент вместе с элементом, который вы делаете. Можно использовать замыкание для этого свойства, чтобы создать связанный элемент и получить его идентификатор, как показано в примере 5.11.

Пример 5.11. Создание связанного элемента в сидере

```
$factory->define(Contact::class, function (Faker\Generator $faker){
    return [
        'name' => 'Lupita Smith',
        'email' => 'lupita@gmail.com',
        'company_id' => function () {
            return factory(App\Company::class)->create()->id;
        },
    ];
});
```

Каждому замыканию передается отдельный параметр, который содержит сгенерированный элемент в форме массива вплоть до этого момента. Это можно использовать по-другому, как показано в примере 5.12.

Пример 5.12. Использование значений из других параметров в сидере

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => 'Lupita Smith',
        'email' => 'lupita@gmail.com',
        'company_id' => function () {
            return factory(App\Company::class)->create()->id;
        },
        'company_size' => function ($contact) {
            // Использует свойство "company_id", сгенерированное выше
            return App\Company::find($contact['company_id'])->size;
        },
    ];
});
```

Определение и доступ к нескольким состояниям фабрики моделей. На секунду вернемся к `ContactFactory.php` (из примеров 5.8 и 5.9). Мы определили базовую фабрику `Contact`:

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});
```

Но иногда вам нужно несколько фабрик для класса объекта. Что, если необходимо добавить несколько контактов очень важных людей (VIP)? Для этого мы можем использовать метод `state()`, чтобы определить второе состояние, как показано в примере 5.13. Первый параметр `state()` по-прежнему является именем создаваемой вами сущности, второй — названием вашего состояния, а третий — массивом любых специально устанавливаемых атрибутов для этого состояния.

Пример 5.13. Определение нескольких состояний фабрики для одной и той же модели

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});

$factory->state(Contact::class, 'vip', [
    'vip' => true,
]);
```

Если для измененных атрибутов требуется больше, чем простое статическое значение, можно передать замыкание вместо массива в качестве второго параметра, а затем вернуть массив атрибутов, которые вы хотите изменить, как в примере 5.14.

Пример 5.14. Указание состояния фабрики с помощью замыкания

```
$factory->state(Contact::class, 'vip', function (Faker\Generator $faker) {
    return [
        'vip' => true,
        'company' => $faker->company,
    ];
});
```

Теперь создадим экземпляр определенного состояния:

```
$vip = factory(Contact::class)->state('vip')->create();

$vips = factory(Contact::class, 3)->state('vip')->create();
```



Состояния фабрики до Laravel 5.3

В проектах, работающих под управлением версий Laravel до 5.3, состояния фабрики назывались типами фабрик и нужно было использовать `$factory->defineAs()` вместо `$factory->state()`. Вы можете узнать об этом больше в документации по 5.2 по адресу <http://bit.ly/2Fmnaew>.

Так много информации. Не беспокойтесь, если возникли трудности — последний раздел определенно был непростой. Вернемся к началу и поговорим об основном компоненте инструментов баз данных Laravel — генераторе запросов.

Генератор запросов

Теперь, когда вы подключены, перенесли и заполнили свои таблицы, начнем разбираться с использованием инструментов базы данных. В основе каждой части функциональности базы данных Laravel лежит генератор запросов — гибкий интерфейс для взаимодействия с несколькими различными типами баз данных с помощью единого понятного API.

ЧТО ТАКОЕ ТЕКУЧИЙ ИНТЕРФЕЙС

Текущий интерфейс (fluent interface) в основном использует цепочку методов, чтобы предоставить конечному пользователю более простой API. Вместо того чтобы ожидать, что все соответствующие данные будут переданы в конструктор или вызов метода, текущие цепочки вызовов могут быть построены постепенно, с последовательными вызовами. Сравним:

```
// Не текущие:
$users = DB::select(['table' => 'users', 'where' => ['type' => 'donor']]);

// Текущие:
$users = DB::table('users')->where('type', 'donor')->get();
```

Архитектура базы данных Laravel позволяет подключаться к MySQL, PostgreSQL, SQLite и SQL Server через единый интерфейс, просто изменив несколько параметров конфигурации.

Если вы когда-либо работали с фреймворком PHP, то могли пользоваться инструментом для запуска «необработанных» SQL-запросов с базовым экранированием ради безопасности. Таков конструктор запросов с множеством удобных слоев и хелперов. Начнем с простых вызовов.

Стандартное использование фасада DB

Прежде чем приступить к построению сложных запросов с помощью связывания методов цепочкой, рассмотрим несколько примеров команд для построения запросов. Фасад DB используется для построения цепочки запросов и более простых необработанных запросов, как показано в примере 5.15.

Пример 5.15. Пример использования необработанного SQL и генератора запросов

```
// Базовое утверждение
DB::statement('drop table users');

// Необработанный выбор и привязка параметров
DB::select('select * from contacts where validated = ?', [true]);

// Выбрать, используя текущий генератор
$users = DB::table('users')->get();

// Соединения и другие сложные вызовы
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.type', 'donor');
    })
    ->get();
```

Чистый SQL

Как вы видели в примере 5.15, можно выполнить любой необработанный вызов базы данных, используя фасад DB и метод `statement(): DB::statement('оператор SQL')`.

Но есть и специальные методы для различных обычных действий: `select()`, `insert()`, `update()` и `delete()`. Это все еще необработанные вызовы, но есть разница. Во-первых, использование `update()` и `delete()` вернет количество затронутых строк, тогда как `statement()` — нет; во-вторых, с помощью этих методов будущим разработчикам станет понятнее, какими именно инструкциями вы пользуетесь.

Необработанные выборки

Простейшим из конкретных методов DB является `select()`. Вы можете запустить его без каких-либо дополнительных параметров:

```
$users = DB::select('select * from users');
```

Это вернет массив объектов `stdClass`.

КОЛЛЕКЦИИ ILLUMINATE

5.3 До Laravel 5.3 фасад DB возвращал объект `stdClass` для методов, которые возвращают только одну строку (таких как `first()`), и массив для любых других, которые возвращают несколько строк (например, `all()`). В Laravel 5.3+ фасад DB, как и Eloquent, возвращает коллекцию для любого метода, который возвращает (или может вернуть) несколько строк. Фасад DB возвращает экземпляр `Illuminate\Support\Collection`, а Eloquent — экземпляр `Illuminate\Database\Eloquent\Collection`, который расширяет `Illuminate\Support\Collection` несколькими методами, специфичными для Eloquent.

Коллекция `Collection` подобна массиву PHP с суперспособностями, позволяя вам запускать `map()`, `filter()`, `redu()`, `each()` и многое другое для ваших данных. Вы можете узнать больше о коллекциях в главе 17.

Привязки параметров и именованные привязки

Архитектура базы данных Laravel позволяет использовать привязку параметров PDO, что защищает ваши запросы от потенциальных SQL-атак. Передать параметр оператору так же просто, как заменить значение в вашем запросе на `?`, а затем добавить значение ко второму параметру вашего вызова:

```
$usersOfType = DB::select(
    'select * from users where type = ?',
    [$type]
);
```

Вы также можете назвать эти параметры для ясности:

```
$usersOfType = DB::select(
    'select * from users where type = :type',
    ['type' => $userType]
);
```

Необработанные вставки

Все необработанные команды выглядят практически одинаково. Необработанные вставки выглядят так:

```
DB::insert(
    'insert into contacts (name, email) values (?, ?)',
    ['sally', 'sally@me.com']
);
```

Необработанные обновления

Обновления выглядят так:

```
$countUpdated = DB::update(
    'update contacts set status = ? where id = ?',
    ['donor', $id]
);
```

Необработанные удаления

А удаления — так:

```
$countDeleted = DB::delete(
    'delete from contacts where archived = ?',
    [true]
);
```

Выстраивание цепочки с генератором запросов

До сих пор мы фактически не использовали конструктор запросов как таковой. Мы выполняли только простые вызовы методов в фасаде **DB**. Построим несколько запросов по-настоящему.

Генератор запросов позволяет объединить методы, чтобы *сгенерировать запрос*. В конце цепочки вы будете использовать какой-то метод — скорее всего, **get()** — для запуска фактического выполнения только что созданного запроса.

Посмотрим на простой пример:

```
$usersOfType = DB::table('users')
    ->where('type', $type)
    ->get();
```

Здесь мы построили наш запрос — таблица `users`, тип `$type`, — затем выполнили запрос и получили результат.

Посмотрим, какие методы позволяет сочетать генератор запросов. Я буду разделять их на ограничивающие, модифицирующие, условные и завершающие/возвращающие.

Ограничивающие методы

Эти методы принимают запрос таким, какой он есть, и ограничивают его так, чтобы он возвращал меньшее подмножество возможных данных.

❑ `select()`. Позволяет вам выбрать столбцы для выборки:

```
$emails = DB::table('contacts')
    ->select('email', 'email2 as second_email')
    ->get();
// Или
$emails = DB::table('contacts')
    ->select('email')
    ->addSelect('email2 as second_email')
    ->get();
```

❑ `where()`. Позволяет ограничить объем того, что возвращается с помощью `WHERE`. По умолчанию сигнатура метода `where()` задает, что он принимает три параметра — столбец, оператор сравнения и значение:

```
$newContacts = DB::table('contact')
    ->where('created_at', '>', now()->subDay())
    ->get();
```

Если ваше сравнение — это `=`, что наиболее распространенный случай, можно опустить второй оператор.

```
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

Если вы хотите использовать несколько операторов `where()`, объедините их в цепочку или передайте массив массивов:

```
$newVips = DB::table('contacts')
    ->where('vip', true)
    ->where('created_at', '>', now()->subDay());
// Или
$newVips = DB::table('contacts')->where([
    ['vip', true],
    ['created_at', '>', now()->subDay()],
]);
```

❑ `orWhere()`. Создает простые операторы `OR WHERE`:

```
$priorityContacts = DB::table('contacts')
    ->where('vip', true)
    ->orWhere('created_at', '>', now()->subDay())
    ->get();
```


Чтобы создать более сложный оператор **OR WHERE** с несколькими условиями, передайте `orWhere()` замыкание:

```
$contacts = DB::table('contacts')
    ->where('vip', true)
    ->orWhere(function ($query) {
        $query->where('created_at', '>', now()->subDay())
            ->where('trial', false);
    })
    ->get();
```



Потенциальная путаница с множественными вызовами `where()` и `orWhere()`

Если вы используете вызовы `orWhere()` вместе с несколькими вызовами `where()`, обязательно убедитесь, что запрос выполняет именно то, что вы хотите. Не из-за какой-либо ошибки в Laravel, а потому, что подобный запрос может не делать того, что вы ожидаете:

```
$canEdit = DB::table('users')
    ->where('admin', true)
    ->orWhere('plan', 'premium')
    ->where('is_plan_owner', true)
    ->get();
```

```
SELECT * FROM users
WHERE admin = 1
OR plan = 'premium'
AND is_plan_owner = 1;
```

Если вы хотите написать SQL, который говорит «если это **ИЛИ** (это и это)», что явно подразумевается в предыдущем примере, нужно передать замыкание в вызов `orWhere()`:

```
$canEdit = DB::table('users')
    ->where('admin', true)
    ->orWhere(function ($query) {
        $query->where('plan', 'premium')
            ->where('is_plan_owner', true);
    })
    ->get();
SELECT * FROM users
WHERE admin = 1
OR (plan = 'premium' AND is_plan_owner = 1);
```

- ❑ `whereBetween(colName, [low, high])`. Позволяет вам сгенерировать запрос так, чтобы он возвращал только те строки, в которых столбец находится между двумя значениями (включая эти два значения):

```
$mediumDrinks = DB::table('drinks')
    ->whereBetween('size', [6, 12])
    ->get();
```

Так же работает `whereNotBetween()`, но с обратным выбором.

- ❑ `whereIn(colName, [1, 2, 3])`. Позволяет вам выполнить запрос, который возвратит только те строки, для которых значение столбца указано в явно предоставленном списке параметров:

```
$closeBy = DB::table('contacts')
->whereIn('state', ['FL', 'GA', 'AL'])
->get();
```

То же самое для `whereNotIn()`, но с обратным результатом.

- ❑ `whereNull(colName)` и `whereNotNull(colName)`. Позволяют вам выбирать только те строки, в которых данный столбец равен `NULL` или `NOT NULL` соответственно.
- ❑ `whereRaw()`. Позволяет передать необработанную, неэкранированную строку, которая будет добавлена после оператора `WHERE`:

```
$goofs = DB::table('contacts')->whereRaw('id = 12345')->get()
```



Остерегайтесь SQL-инъекций!

Любые запросы SQL, передаваемые в `whereRaw()`, не будут экранированы. Используйте этот метод осторожно и как можно реже. Это отличная возможность для атак с применением SQL-инъекций в вашем приложении.

- ❑ `whereExists()`. Позволяет выбрать только те строки, которые при передаче в предоставленный подзапрос возвращают хотя бы одну строку. Представьте, что вы хотите получить имена только тех пользователей, которые оставили хотя бы один комментарий:

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

- ❑ `distinct()`. Выбирает только те строки, в которых выбранные данные уникальны по сравнению с другими в возвращаемых данных. Обычно это используется в сочетании с `select()`, потому что, если вы применяете первичный ключ, дублированных строк не будет:

```
$lastNames = DB::table('contacts')->select('city')->distinct()->get();
```

Модифицирующие методы

Эти методы изменяют способ вывода результатов запроса, а не просто ограничивают его результаты.

- ❑ `orderBy(colName, direction)`. Сортирует результаты. Второй параметр может быть `asc` (по умолчанию в порядке возрастания) или `desc` (в порядке убывания):

```
$contacts = DB::table('contacts')
->orderBy('last_name', 'asc')
->get();
```

- ❑ `groupBy()` и `having()` или `havingRaw()`. Группирует ваши результаты по столбцу. К тому же `having()` и `havingRaw()` дают возможность фильтровать результаты на основе свойств групп. Например, вы можете искать только те города, в которых проживает минимум 30 человек:

```
$populousCities = DB::table('contacts')
->groupBy('city')
->havingRaw('count(contact_id) > 30')
->get();
```

- ❑ `skip()` и `take()`. Чаще всего они используются для разбивки на страницы (пагинации), позволяя задать, сколько строк нужно вернуть и пропустить до начала возврата, например номер и размер страницы в системе разбивки на страницы:

```
// возвращает строки 31-40
$page4 = DB::table('contacts')->skip(30)->take(10)->get();
```

- ❑ `latest(colName)` и `oldest(colName)`. Сортировать по переданному столбцу (или `created_at`, если имя столбца не передано) в порядке убывания (`latest()`) или возрастания (`oldest()`).
- ❑ `inRandomOrder()`. Сортирует результат случайным образом.

Условные методы

Существует два метода в Laravel 5.2 и более поздних версиях, позволяющих условно применять их «содержимое» (отправляемое замыкание) на основе логического состояния передаваемого значения.

- ❑ `when()`. Если задан верный первый параметр, применяется модификация запроса, содержащаяся в замыкании. Если первый параметр ложный, то ничего не происходит. Обратите внимание, что он может быть логическим (например, `$ignoreDrafts`, установленным в `true` или `false`), необязательным значением (`$status`, извлекается из ввода пользователя и по умолчанию имеет значение `null`) или замыканием, которое возвращает что-либо из них. Важно: оно оценивается как правдивое или ложное. Например:

```
$status = request('status'); // По умолчанию null, если не установлено

$posts = DB::table('posts')
->when($status, function ($query) use ($status) {
```

```
        return $query->where('status', $status);
    })
    ->get();

// Или
$post = DB::table('posts')
    ->when($ignoreDrafts, function ($query) {
        return $query->where('draft', false);
    })
    ->get();
```

Вы также можете передать третий параметр, другое замыкание, которое будет применяться при неверности первого параметра.

- ❑ **unless()**. Точная инверсия **when()**. Если первый параметр ложный, запускается второе замыкание.

Завершающие/возвращающие методы

Эти методы останавливают цепочку запросов и запускают выполнение запроса SQL. Без одного из них в конце цепочки запросов всегда будет возвращаться просто экземпляр генератора запросов. Добавьте любой из этих методов в генератор запросов, и вы получите результат.

- ❑ **get()**. Получает все результаты построенного запроса:

```
$contacts = DB::table('contacts')->get();
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

- ❑ **first()** и **firstOrFail()**. Получает только первый результат — то же, что и **get()**, но с добавленным **LIMIT 1**:

```
$newestContact = DB::table('contacts')
    ->orderBy('created_at', 'desc')
    ->first();
```

first() завершается неудачей, если результатов нет, тогда как **firstOrFail()** генерирует исключение.

Если вы передадите массив имен столбцов одному из этих методов, он вернет данные только для этих столбцов.

- ❑ **find(id)** и **findOrFail(id)**. Подобно **first()**, но вы передаете значение идентификатора, соответствующее первичному ключу для поиска. **find()** тихо завершается неудачей, если строки с таким идентификатором не существует, а **findOrFail()** выдает исключение:

```
$contactFive = DB::table('contacts')->find(5);
```

- ❑ **value()**. Выбирает значение только из одного поля из первой строки. Подобно **first()**, но для случая, если вам нужен только определенный столбец:

```
$newestContactEmail = DB::table('contacts')
    ->orderBy('created_at', 'desc')
    ->value('email');
```

- ❑ `count()`. Возвращает целое — количество всех соответствующих результатов:

```
$countVips = DB::table('contacts')
    ->where('vip', true)
    ->count();
```

- ❑ `min()` и `max()`. Вернуть минимальное или максимальное значение в определенном столбце:

```
$highestCost = DB::table('orders')->max('amount');
```

- ❑ `sum()` и `avg()`. Вернуть сумму или среднее значение всех значений в определенном столбце:

```
$averageCost = DB::table('orders')
    ->where('status', 'completed')
    ->avg('amount');
```

Написание необработанных запросов внутри методов генератора запросов с помощью `DB::raw`

Вы уже видели несколько пользовательских методов для необработанных операторов — например, `select()` имеет аналог `selectRaw()`, позволяющий передать строку, которую генератор запросов должен поместить после оператора `WHERE`.

Однако вы также можете передать результат вызова `DB::raw()` практически любому методу в генераторе запросов для того же результата:

```
$contacts = DB::table('contacts')
    ->select(DB::raw('*', (score * 100) AS integer_score'))
    ->get();
```

Соединения

Иногда соединения (`join`) трудно определить, и существует множество возможностей фреймворка для их упрощения, но генератор запросов сделает все сам наилучшим образом. Рассмотрим пример:

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->select('users.*', 'contacts.name', 'contacts.status')
    ->get();
```

Метод `join()` создает внутреннее соединение (`inner join`). Можно объединить несколько соединений одно за другим или использовать `leftJoin()`, чтобы получить соединение слева (`left join`).

Вы можете создавать более сложные соединения, передавая замыкание в метод `join()`:

```
DB::table('users')
->join('contacts', function ($join) {
    $join
        ->on('users.id', '=', 'contacts.user_id')
        ->orOn('users.id', '=', 'contacts.proxy_user_id');
})
->get();
```

Объединения

Возможна совокупность двух запросов (объединить их результаты в один набор), сначала создав их, а затем используя метод `union()` или `unionAll()`:

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

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

Вставки

Метод `insert()` довольно прост. Передайте ему массив для вставки одной строки или массив массивов для вставки нескольких строк и используйте `insertGetId()` вместо `insert()`, чтобы получить идентификатор первичного ключа с автоинкрементом в качестве возврата:

```
$id = DB::table('contacts')->insertGetId([
    'name' => 'Abe Thomas',
    'email' => 'athomas1987@gmail.com',
]);

DB::table('contacts')->insert([
    ['name' => 'Tamika Johnson', 'email' => 'tamikaj@gmail.com'],
    ['name' => 'Jim Patterson', 'email' => 'james.patterson@hotmail.com'],
]);
```

Обновления

Обновления тоже просты. Создайте свой запрос на обновление, вместо `get()` или `first()` используйте `update()` и передайте ему массив параметров:

```
DB::table('contacts')
->where('points', '>', 100)
->update(['status' => 'vip']);
```

Вы также можете использовать быстрый инкремент/декремент для столбцов, применяя методы `increment()` и `decrement()`. Первый параметр каждого метода — это имя столбца, а второй (необязательный) — число, на которое нужно произвести инкремент/декремент:

```
DB::table('contacts')->increment('tokens', 5);
DB::table('contacts')->decrement('tokens');
```

Удаления

Удаления еще проще. Создайте свой запрос, а затем завершите его с помощью `delete()`:

```
DB::table('users')
    ->where('last_login', '<', now()->subYear())
    ->delete();
```

Можно усесть таблицу, удалив каждую строку с одновременным сбросом идентификатора автоинкремента:

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

Операции JSON

При наличии столбцов JSON можно обновить или выбрать строки на основе аспектов структуры JSON, используя синтаксис стрелки для обхода дочерних элементов:

```
// Выбрать все записи, где свойство "isAdmin" столбца "options"
// JSON установлено в true
DB::table('users')->where('options->isAdmin', true)->get();
```

```
// Обновить все записи, устанавливая свойство "verified"
// столбца "options" JSON в true
DB::table('users')->update(['options->isVerified', true]);
```

5.3 Это новая функция, начиная с Laravel 5.3.

Транзакции

Транзакции позволяют скомпоновать серию запросов к базе данных так, чтобы они выполнялись в пакете, который можно откатить, отменяя всю серию запросов. Транзакции часто используются для гарантии того, что *все* или *никакие*, но не *некоторые* из серии связанных запросов выполняются — в случае сбоя ORM откатит всю серию запросов.

Если в какой-либо точке выполнения транзакции возникают какие-либо исключения, все запросы в транзакции будут откатываться. Если закрытие транзакции завершится успешно, все запросы подтвердятся и не откатятся.

Посмотрим на транзакцию в примере 5.16.

Пример 5.16. Простая транзакция базы данных

```
DB::transaction(function () use ($userId, $numVotes) {  
    // Возможно, неудача запроса к DB  
    DB::table('users')  
        ->where('id', $userId)  
        ->update(['votes' => $numVotes]);  
  
    // Кэшируем запрос, который не хотим запускать, если вышеуказанный запрос не удался  
    DB::table('votes')  
        ->where('user_id', $userId)  
        ->delete();  
});
```

Можно предположить, что у нас был какой-то предыдущий процесс, который суммировал количество голосов из таблицы `votes` по данному пользователю. Мы хотим кэшировать это число в таблице `users`, а затем убрать эти голоса из таблицы `votes`. Но мы не хотим стирать голоса, *пока* обновление таблицы `users` не выполнится успешно. И не хотим сохранять обновленное количество голосов в таблице `users`, если удаление в `votes` не удастся.

Если что-то пойдет не так с одним запросом, другой не выполнится. Это магия транзакций базы данных.

Вы также можете вручную начинать и завершать транзакции — это относится к запросам генератора запросов и запросам Eloquent. Начните с `DB::beginTransaction()`, закончите с `DB::commit()` и задайте отмену с помощью `DB::rollback()`:

```
DB::beginTransaction();  
  
// Выполнить действия с базой данных  
  
if ($badThingsHappened) {  
    DB::rollback();  
}  
  
// Выполнить другие действия с базой данных  
  
DB::commit();
```

Введение в Eloquent

Теперь поговорим об Eloquent — флагманском инструменте базы данных Laravel, который основан на генераторе запросов.

Eloquent представляет собой ActiveRecord ORM — уровень абстракции базы данных, который обеспечивает единый интерфейс для взаимодействия с несколькими типами баз данных. ActiveRecord означает, что один класс Eloquent отвечает

не только за предоставление возможности взаимодействовать с таблицей в целом (например, `User::all()` получит всех пользователей), но также за представление отдельной строки таблицы (скажем, `$sharon = new User`). Кроме того, каждый экземпляр способен управлять своим собственным существованием; вы можете вызвать `$sharon->save()` или `$sharon->delete()`.

Eloquent фокусируется на простоте и, как и остальная часть фреймворка, опирается на «соглашение о конфигурации», позволяющее создавать мощные модели с минимумом кода.

Например, вы можете выполнить все операции в примере 5.18 с моделью, определенной в примере 5.17.

Пример 5.17. Простейшая модель Eloquent

```
<?php

use Illuminate/Database/Eloquent/Model;

class Contact extends Model {}
```

Пример 5.18. Операции, возможные с помощью простейшей модели Eloquent

```
// В контроллере
public function save(Request $request)
{
    // Создать и сохранить новый контакт из ввода пользователя
    $contact = new Contact();
    $contact->first_name = $request->input('first_name');
    $contact->last_name = $request->input('last_name');
    $contact->email = $request->input('email');
    $contact->save();

    return redirect('contacts');
}

public function show($contactId)
{
    // Возвращаем JSON-представление контакта на основе сегмента URL;
    // если контакт не существует, выдается исключение
    return Contact::findOrFail($contactId);
}

public function vips()
{
    // Неоправданно сложный пример, но все еще возможный с базовым классом Eloquent;
    // добавляет свойство formalName к каждой записи VIP
    return Contact::where('vip', true)->get()->map(function ($contact) {
        $contact->formalName = "The exalted {$contact->first_name} of the
            {$contact->last_name}s";

        return $contact;
    });
}
```

Как же это обеспечивается? Eloquent угадывает имя таблицы (`Contact` становится `contacts`), и так у вас появляется полнофункциональная модель Eloquent.

Рассмотрим работу с моделями Eloquent.

Создание и определение моделей Eloquent

Во-первых, создадим модель. Для этого есть команда Artisan:

```
php artisan make:model Contact
```

Вот что мы получим в `app/Contact.php`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```



Создание миграции вместе с вашей моделью

Если вы хотите автоматически выполнить миграцию при создании модели, передайте флаг `-m` или `--migration`:

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

Имя таблицы

Поведение по умолчанию для имен таблиц заключается в том, что Laravel применяет «змеиный регистр» (snake case) и множественно использует название вашего класса, поэтому `SecondaryContact` получит доступ к таблице с именем `secondary_contacts`. Если вы хотите явно задать имя, задайте свойство `$table` для модели:

```
protected $table = 'contacts_secondary';
```

Первичный ключ

По умолчанию Laravel предполагает, что у каждой таблицы будет первичный ключ под названием `id` с автоинкрементным целым числом.

Если вы хотите изменить имя вашего первичного ключа, поменяйте свойство `$primaryKey`:

```
protected $primaryKey = 'contact_id';
```

Если вы хотите, чтобы он был неинкрементным, используйте:

```
public $incrementing = false;
```

Метки времени

Eloquent ожидает, что в каждой таблице будут столбцы меток времени `created_at` и `updated_at`. Если в вашей таблице их нет, отключите функциональность `$timestamps`:

```
public $timestamps = false;
```

Вы можете настроить формат, который Eloquent использует для хранения ваших временных меток в базе данных, установив для свойства класса `$dateFormat` пользовательскую строку. Строка будет проанализирована синтаксисом РНР-функции `date()`, поэтому в следующем примере дата будет храниться в секундах с начала эпохи Unix:

```
protected $dateFormat = 'U';
```

Получение данных с помощью Eloquent

В большинстве случаев при извлечении данных из своей БД с помощью Eloquent вы используете статические вызовы в своей модели Eloquent.

Начнем с получения всего:

```
$allContacts = Contact::all();
```

Это было легко. Немного отфильтруем:

```
$vipContacts = Contact::where('vip', true)->get();
```

Фасад Eloquent дает возможность располагать ограничения цепочкой, теперь они выглядят знакомо:

```
$newestContacts = Contact::orderBy('created_at', 'desc')
    ->take(10)
    ->get();
```

Как только вы оказались за начальным именем фасада, вы просто работаете с генератором запросов Laravel. Все действия с помощью генератора запросов на фасаде DB можно выполнить и с вашими объектами Eloquent.

Получение одного результата

Как мы уже говорили, вы можете использовать `first()` для возвращения только первой записи из запроса, или `find()`, чтобы получить только запись по указанному идентификатору. Оба метода, если вы добавите `OrFail` к имени метода, выдаст исключение, если не найдено соответствующих результатов. Поэтому `findOrFail()` — распространенный инструмент для поиска сущности по сегменту

URL (или выдачи исключения, если соответствующей сущности нет), как видно в примере 5.19.

Пример 5.19. Использование метода Eloquent `OrFail()` в методе контроллера

```
// ContactController
public function show($contactId)
{
    return view('contacts.show')
        ->with('contact', Contact::findOrFail($contactId));
}
```

Любой метод, предназначенный для возврата одной записи (`first()`, `firstOrFail()`, `find()` или `findOrFail()`), возвращает экземпляр класса Eloquent. Таким образом, `Contact::first()` вернет экземпляр класса `Contact`, заполненный данными из первой строки таблицы.



Исключения

Как вы можете видеть в примере 5.19, нам не нужно отлавливать исключение Eloquent «модель не найдена» (`Illuminate\Database\Eloquent\ModelNotFoundException`) в наших контроллерах. Система маршрутизации Laravel отловит его и выдаст для нас ошибку 404.

Конечно, вы можете отловить это конкретное исключение и обработать его, если хотите.

Получение нескольких результатов

`get()` работает с Eloquent так же, как и в обычных вызовах генератора запросов, — создайте запрос и в конце вызовите `get()`, чтобы получить результаты:

```
$vipContacts = Contact::where('vip', true)->get();
```

Однако есть специфичный для Eloquent метод `all()`, который, как вы увидите, разработчики используют для получения неотфильтрованного списка всех данных в таблице:

```
$contacts = Contact::all();
```



Использование `get()` вместо `all()`

Каждый раз, когда вы применяете `all()`, можно использовать `get()`. `Contact::get()` даст тот же ответ, что и `Contact::all()`. Однако, как только вы начнете изменять запрос, например добавив фильтр `where()`, `all()` больше не будет работать, но `get()` — продолжит.

Так что, хотя `all()` очень распространен, я бы рекомендовал во всех случаях использовать `get()` и игнорировать существование `all()`.

 Другая особенность метода `get()` в Eloquent (в отличие от `all()`) в том, что до Laravel 5.3 он возвращал массив моделей вместо коллекции. В 5.3 и поздних они оба возвращают коллекции.

Разбиение ответов на порции с помощью `chunk()`

Если вам когда-либо требовалось обрабатывать большое количество (тысячи или более) записей одновременно, могли возникнуть проблемы с памятью или блокировкой. Laravel позволяет разбивать ваши запросы на более мелкие части (порции) и обрабатывать их в пакетном режиме, сохраняя маленький объем памяти вашего большого запроса. Пример 5.20 иллюстрирует использование `chunk()` для разбиения запроса на «порции» по 100 записей в каждой.

Пример 5.20. Разделение запроса Eloquent для ограничения использования памяти

```
Contact::chunk(100, function ($contacts) {  
    foreach ($contacts as $contact) {  
        // Действия с $contact  
    }  
});
```

Агрегаты

Агрегаты в генераторе запросов доступны и для запросов Eloquent. Например:

```
$countVips = Contact::where('vip', true)->count();  
$sumVotes = Contact::sum('votes');  
$averageSkill = User::avg('skill_level');
```

Вставки и обновления с помощью Eloquent

Вставка и обновление значений — одно из мест, где Eloquent начинает расходиться с обычным синтаксисом генератора запросов.

Вставки

Есть два основных способа вставить новую запись с помощью Eloquent.

Можно создать новый экземпляр вашего класса Eloquent, установить нужные свойства вручную и вызвать `save()` для этого экземпляра, как в примере 5.21.

Пример 5.21. Вставка записи Eloquent путем создания нового экземпляра

```
$contact = new Contact;  
$contact->name = 'Ken Hirata';
```

```
$contact->email = 'ken@hirata.com';  
$contact->save();
```

```
// или
```

```
$contact = new Contact([  
    'name' => 'Ken Hirata',  
    'email' => 'ken@hirata.com',  
]);  
$contact->save();
```

```
// или
```

```
$contact = Contact::make([  
    'name' => 'Ken Hirata',  
    'email' => 'ken@hirata.com',  
]);  
$contact->save();
```

До вызова `save()` этот экземпляр `Contact` полностью представляет контакт, за исключением того, что он никогда не сохранялся в базе данных. Значит, у него нет идентификатора. Если приложение завершит работу, он не сохранится и для него не будут установлены значения `created_at` и `updated_at`.

Вы также можете передать массив в `Model::create()`, как показано в примере 5.22. В отличие от `make()`, `create()` сохраняет экземпляр в базе данных, как только его вызвали.

Пример 5.22. Вставка записи Eloquent путем передачи массива в `create()`

```
$contact = Contact::create([  
    'name' => 'Keahi Hale',  
    'email' => 'halek481@yahoo.com',  
]);
```

В любом контексте, где вы передаете массив (в `new Model()`, `Model::make()`, `Model::create()` или `Model::update()`), каждое свойство, устанавливаемое через `Model::create()`, должно быть одобрено для «массового назначения», о котором я вскоре расскажу. Но это не требуется для первого варианта в примере 5.21, где вы назначаете каждое свойство индивидуально.

Обратите внимание: если вы используете `Model::create()`, не нужно делать `save()` для экземпляра — это часть метода `create()` модели.

Обновления

Обновление записей выглядит очень похоже на вставку. Можно получить конкретный экземпляр, изменить его свойства и сохранить. Или сделать один вызов и передать массив обновленных свойств. Пример 5.23 иллюстрирует первый подход.

Пример 5.23. Обновление записи Eloquent путем обновления экземпляра и сохранения

```
$contact = Contact::find(1);  
$contact->email = 'natalie@parkfamily.com';  
$contact->save();
```

Поскольку эта запись уже существует, у нее будет метка времени `created_at` и прежний идентификатор, но поле `updated_at` изменится на текущую дату и время. Пример 5.24 иллюстрирует второй подход.

Пример 5.24. Обновление одной или нескольких записей Eloquent путем передачи массива в метод `update()`

```
Contact::where('created_at', '<', now()->subYear())  
    ->update(['longevity' => 'ancient']);
```

// или

```
$contact = Contact::find(1);  
$contact->update(['longevity' => 'ancient']);
```

Этот метод ожидает массив, где каждый ключ — имя столбца, а каждое значение — значение столбца.

Массовое назначение

Мы рассмотрели несколько примеров передачи массивов значений в методы класса Eloquent. Однако ни один из них на самом деле не будет работать, пока вы не определите, какие поля «способны к заполнению» в модели.

Цель этого — защитить вас от (возможно, злонамеренного) пользовательского ввода, случайно устанавливающего новые значения в полях, которые вы не хотите изменять. Рассмотрим общий сценарий в примере 5.25.

Пример 5.25. Обновление модели Eloquent с использованием всего ввода запроса

```
// ContactController  
public function update(Contact $contact, Request $request)  
{  
    $contact->update($request->all());  
}
```

Если вы не знакомы с объектом `Request Illuminate`: в примере 5.25 будет приниматься каждый фрагмент пользовательского ввода и передаваться в метод `update()`. Показанный метод `all()` включает в себя параметры URL и входные данные формы, поэтому злоумышленник может легко добавить туда что-то вроде `id` и `owner_id`, которые вы, вероятно, не хотите обновлять.

К счастью, это не сработает, пока вы не определите поля, которые могут быть заполнены в вашей модели. Можно внести в белый список поля для заполнения

либо внести в черный список «защищенные» поля, чтобы определить, какие поля можно или нельзя редактировать с помощью «массового назначения», то есть путем передачи массива значений в `create()` или `update()`. Обратите внимание, что незаполняемые свойства все равно могут быть изменены прямым присвоением (например, `$contact->password = 'abc';`). В примере 5.26 показаны оба подхода.

Пример 5.26. Использование заполняемых или защищенных свойств Eloquent для определения полей, открытых для массового заполнения

```
class Contact
{
    protected $fillable = ['name', 'email'];

    // или

    protected $guarded = ['id', 'created_at', 'updated_at', 'owner_id'];
}
```



Использование `Request::only()` с массовым заполнением Eloquent

В примере 5.25 нам была нужна защита массового назначения Eloquent, потому что мы использовали метод `all()` объекта `Request` для передачи всего пользовательского ввода.

Защита массовых назначений Eloquent — отличный инструмент, но есть и полезный прием, который не дает вам воспринимать какие-либо старые данные из пользовательского ввода.

Класс `Request` содержит метод `only()`, позволяющий выдернуть только несколько ключей из пользовательского ввода. Теперь вы можете сделать так:

```
Contact::create($request->only('name', 'email'));
```

`firstOrCreate()` и `firstOrCreateNew()`

Иногда вы хотите сказать своему приложению: «Дайте мне экземпляр с такими-то свойствами или, если он не существует, создайте его». Именно здесь появляются методы `firstOrCreate()`.

Методы `firstOrCreate()` и `firstOrCreateNew()` принимают массив ключей и значений в качестве первого параметра:

```
$contact = Contact::firstOrCreate(['email' => 'luis.ramos@myacme.com']);
```

Они будут искать и извлекать первую запись, соответствующую этим параметрам, и, если нет соответствующих записей, создадут экземпляр с такими свойствами; `firstOrCreate()` сохранит этот экземпляр в базе данных, а затем вернет, а `firstOrCreateNew()` вернет его без сохранения.

Если вы передадите массив значений в качестве второго параметра, значения будут добавлены в созданную запись (если она была создана), но не будут использоваться для поиска и выяснения существования записи.

Удаление с помощью Eloquent

Удаление с помощью Eloquent очень похоже на обновление посредством Eloquent, но с необязательным мягким удалением вы можете архивировать стертые элементы для последующей проверки или даже восстановления.

Нормальные удаления

Самый простой способ удалить запись модели — это вызвать метод `delete()` самого экземпляра:

```
$contact = Contact::find(5);  
$contact->delete();
```

Однако если у вас есть только идентификатор, не надо искать экземпляр, чтобы просто удалить его. Можно передать идентификатор или массив идентификаторов методу `destroy()` модели, чтобы удалить их напрямую:

```
Contact::destroy(1);  
// или  
Contact::destroy([1, 5, 7]);
```

Теперь вы можете удалить все результаты запроса:

```
Contact::where('updated_at', '<', now()->subYear())->delete();
```

Мягкие удаления

Мягкие удаления помечают строки базы данных как стертые без фактического удаления их из базы данных. Так можно просматривать их позже, узнать о записи больше, чем просто «нет информации, удалено» при отображении исходной информации, и иметь возможность для ваших пользователей (или администраторов) восстановить некоторые или все данные.

Сложность ручного кодирования приложения с мягкими удалениями в том, что *каждый запрос*, который вы когда-либо напишете, должен будет исключать мягко удаленные данные из поиска. Если вы используете мягкое удаление Eloquent, каждый ваш запрос будет игнорировать их по умолчанию, если только вы явно не скажете вернуть их обратно.

Функция мягкого удаления Eloquent требует добавления в таблицу столбца `deleted_at`. Как только вы включите мягкое удаление в этой модели Eloquent,

каждый написанный вами запрос (если вы явно не включите мягко удаленные записи) будет ограничен так, чтобы игнорировать мягко удаленные строки.

КОГДА СЛЕДУЕТ ИСПОЛЬЗОВАТЬ МЯГКИЕ УДАЛЕНИЯ

Такая возможность не означает, что вы всегда должны ее использовать. Многие люди в сообществе Laravel по умолчанию применяют мягкое удаление в каждом проекте только потому, что функция есть. Однако так существуют реальные затраты. Вполне вероятно, что при просмотре своей базы данных непосредственно с помощью Sequel Pro вы забудете хотя бы один раз проверить столбец `deleted_at`. И если вы не очистите старые удаленные записи, ваши БД будут увеличиваться.

Моя рекомендация: не используйте мягкие удаления по умолчанию. Применяйте при необходимости и очищайте старые мягкие удаления настолько активно, насколько это возможно, с помощью Quicksand (<https://github.com/tightenco/quicksand>). Функция мягкого удаления — мощный инструмент, но его не стоит использовать просто так.

Включение мягких удалений. Чтобы включить мягкое удаление, нужно вставить столбец `deleted_at` в миграции, импортировать признак `SoftDeletes` в модель и добавить `deleted_at` в свойство `$dates`. В генераторе схемы есть метод `softDeletes()`, позволяющий включить столбец `deleted_at` в таблицу, как вы можете видеть в примере 5.27. Пример 5.28 показывает модель Eloquent с включенным мягким удалением.

Пример 5.27. Миграция для добавления столбца мягкого удаления в таблицу

```
Schema::table('contacts', function (Blueprint $table) {
    $table->softDeletes();
});
```

Пример 5.28. Модель Eloquent с включенными мягкими удалениями

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Contact extends Model
{
    use SoftDeletes; // использовать признак

    protected $dates = ['deleted_at']; // отметить этот столбец как дату
}
```

После того как вы внесете эти изменения, каждый вызов `delete()` и `destroy()` теперь будет устанавливать в столбце `deleted_at` вашей строки текущую дату

и время вместо фактического удаления этой строки. И все будущие запросы будут исключать эту строку из результата.

Запросы с мягкими удалениями. Итак, как мы можем получить мягко удаленные элементы?

Можно добавить мягко удаленные элементы в запрос:

```
$allHistoricContacts = Contact::withTrashed()->get();
```

Далее воспользоваться методом `trashed()`, чтобы посмотреть, был ли конкретный экземпляр удален мягко:

```
if ($contact->trashed()) {  
    // сделать что-нибудь  
}
```

И вы можете получить только мягко удаленные элементы:

```
$deletedContacts = Contact::onlyTrashed()->get();
```

Восстановление мягко удаленных объектов. Если вы хотите восстановить мягко удаленный элемент, запустите `restore()` в экземпляре или запросе:

```
$contact->restore();
```

```
// или
```

```
Contact::onlyTrashed()->where('vip', true)->restore();
```

Принудительное удаление мягко удаленных объектов. Вы можете полностью удалить мягко удаленный объект, вызвав `forceDelete()` для объекта или в запросе:

```
$contact->forceDelete();
```

```
// или
```

```
Contact::onlyTrashed()->forceDelete();
```

Области видимости

Мы рассмотрели «отфильтрованные» запросы, для которых не просто возвращаем каждый результат из таблицы. Но до сих пор мы писали все вручную, используя генератор запросов.

Локальные и глобальные области действия в Eloquent позволяют определять предварительно созданные «области» (фильтры), которые вы можете использовать каждый раз, когда запрашивается модель («глобально») либо когда вы запрашиваете ее с помощью определенной цепочки методов («локально»).

Локальные области видимости

Локальные области действия наиболее просты для понимания. Рассмотрим такой пример:

```
$activeVips = Contact::where('vip', true)->where('trial', false)->get();
```

Утомительно писать эту комбинацию методов запроса снова и снова. Кроме того, информация о том, как определить того, кто является «активным VIP», теперь распределена по нашему приложению. Нужно централизовать эти знания. Что, если бы мы могли просто написать так?

```
$activeVips = Contact::activeVips()->get();
```

Мы можем — это локальная область действия. И ее легко определить в классе `Contact`, как вы можете видеть в примере 5.29.

Пример 5.29. Определение локальной области в модели

```
class Contact
{
    public function scopeActiveVips($query)
    {
        return $query->where('vip', true)->where('trial', false);
    }
}
```

Чтобы определить локальную область действия, мы добавляем метод в класс Eloquent (начинается со слова `scope`), соединенный с вариантом имени области с большой буквы. Он передается генератору запросов и должен возвращать его, но вы можете изменить запрос перед возвратом — в этом весь смысл.

Можно определить области действия, которые принимают параметры, как показано в примере 5.30.

Пример 5.30. Передача параметров в области действия

```
class Contact
{
    public function scopeStatus($query, $status)
    {
        return $query->where('status', $status);
    }
}
```

И вы используете их таким же образом, просто передавая параметр в область действия:

```
$friends = Contact::status('friend')->get();
```

Глобальные области видимости

Мы говорили, что мягкие удаления работают только в том случае, когда вы выполняете *каждый запрос* в модели, чтобы игнорировать мягко удаленные элементы.

Это глобальная область действия. И мы можем определить наши собственные глобальные области, которые будут применяться к каждому запросу, сделанному из данной модели.

Существует два способа определения глобальной области действия: использование замыкания или целого класса. В каждом из них вы будете регистрировать определенную область в методе `boot()` модели. Начнем с метода замыкания, показанного в примере 5.31.

Пример 5.31. Добавление глобальной области действия с помощью замыкания

```
...
class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('active', function (Builder $builder)
            { $builder->where('active', true);
        });
    }
}
```

Мы только что добавили глобальную область с именем `active`, и теперь каждый запрос в этой модели будет ограничен лишь строками со значением `active true`.

Попробуем более длинный путь, как показано в примере 5.32. Создайте класс, который реализует `Illuminate\Database\Eloquent\Scope`, у него будет метод `apply()`, который принимает экземпляр генератора запросов и модели.

Пример 5.32. Создание класса глобальной области действия

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class ActiveScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        return $builder->where('active', true);
    }
}
```

Чтобы применить эту область действия к модели, еще раз переопределите родительский метод `boot()` и вызовите `addGlobalScope()` для класса, используя `static`, как показано в примере 5.33.

Пример 5.33. Применение глобальной области действия на основе класса

```
<?php

use App\Scopes\ActiveScope;
use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();
        static::addGlobalScope(new ActiveScope);
    }
}
```



Класс Contact без пространства имен

В некоторых из этих примеров использовался класс Contact без пространства имен. Это ненормально, и я просто сэкономил место в книге. Скорее всего, даже ваши модели верхнего уровня будут жить где-то вроде App\Contact.

Удаление глобальных областей видимости. Есть три способа удалить глобальную область, и все используют методы `withoutGlobalScope()` или `withoutGlobalScopes()`. Если вы удаляете область действия на основе замыканий, то первым параметром регистрации `addGlobalScope()` этой области будет ключ, который вы использовали для включения:

```
$allContacts = Contact::withoutGlobalScope('active')->get();
```

Если вы удаляете одну глобальную область действия на основе класса, вы можете передать имя класса в `withoutGlobalScope()` или `withoutGlobalScopes()`:

```
Contact::withoutGlobalScope(ActiveScope::class)->get();
```

```
Contact::withoutGlobalScopes([ActiveScope::class, VipScope::class])->get();
```

Или вы можете просто отключить все глобальные области действия для запроса:

```
Contact::withoutGlobalScopes()->get();
```

Настройка взаимодействия полей с аксессуарами, мутаторами и приведением атрибутов

Теперь, зная, как добавлять и извлекать записи в БД с помощью Eloquent, поговорим о декорировании моделей Eloquent атрибутами и работе с этими атрибутами.

Благодаря использованию аксессуаров (accessors) и мутаторов (mutators) атрибутов, а также их преобразованию можно настраивать способ ввода или вывода отдельных

атрибутов экземпляров Eloquent. Без использования какого-либо из них каждый атрибут вашего экземпляра Eloquent рассматривается как строка. В моделях не может быть таких атрибутов, которых нет в базе данных. Но мы можем это изменить.

Аксессоры

Аксессоры позволяют определять пользовательские атрибуты в ваших моделях Eloquent, когда вы *считываете* данные из экземпляра модели. Этим можно воспользоваться, если вы хотите изменить способ вывода определенного столбца или чтобы создать пользовательский атрибут, которого вообще нет в таблице базы данных.

Определите аксессор, написав метод для вашей модели со следующей структурой: `get{PascalCasedPropertyName}Attribute`. Таким образом, если имя вашего свойства `first_name`, метод аксессора будет называться `getFirstNameAttribute`.

Сначала декорируем ранее существовавший столбец (пример 5.34).

Пример 5.34. Декорирование существующего столбца с использованием аксессоров Eloquent

```
// Определение модели:
class Contact extends Model
{
    public function getNameAttribute($value)
    {
        return $value ? : '(No name provided)';
    }
}

// Использование аксессора:
$name = $contact->name;
```

Можно использовать аксессоры для определения атрибутов, которые никогда не существовали в базе данных, как показано в примере 5.35.

Пример 5.35. Определение атрибута без существующего столбца, используя аксессоры Eloquent

```
// Определение модели:
class Contact extends Model
{
    public function getFullNameAttribute()
    {
        return $this->first_name . ' ' . $this->last_name;
    }
}

// Использование аксессора:
$fullName = $contact->full_name;
```

Мутаторы

Мутаторы работают так же, как аксессоры, но они определяют, как обрабатывать *запись* данных, а не *считывать* их. Можно также использовать мутаторы для изменения процесса записи данных в существующие столбцы или установки столбцов, которых нет в базе данных.

Мутатор можно определить, написав метод для вашей модели со структурой `set{PascalCasedPropertyName}Attribute`. Таким образом, если имя вашего свойства `first_name`, метод мутатора будет называться `setFirstNameAttribute`.

Сначала мы добавим ограничение на обновление существующего столбца (пример 5.36).

Пример 5.36. Декорирование установки значения атрибута с использованием мутаторов Eloquent

```
// Определение мутатора
class Order extends Model
{
    public function setAmountAttribute($value)
    {
        $this->attributes['amount'] = $value > 0 ? $value : 0;
    }
}

// Использование мутатора
$order->amount = '15';
```

Мутаторы будут устанавливать данные в `$this->attributes` с именем столбца в качестве ключа.

Теперь добавим столбец прокси для установки, как показано в примере 5.37.

Пример 5.37. Разрешение для установки значения несуществующего атрибута, используя мутаторы Eloquent

```
// Определение мутатора
class Order extends Model
{
    public function setWorkgroupNameAttribute($workgroupName)
    {
        $this->attributes['email'] = "{$workgroupName}@ourcompany.com";
    }
}

// Использование мутатора
$order->workgroup_name = 'jstott';
```


Редко когда приходится создавать мутатор для несуществующего столбца, ведь может быть непонятно, что нужно задать одно свойство, чтобы оно изменяло другой столбец, но это возможно.

Преобразование атрибутов

Только представьте, сколько времени может потребовать написание аксессоров для преобразования всех ваших полей целочисленного типа в целые числа, кодирования и декодирования JSON для хранения в столбце TEXT или преобразования TINYINT 0 и 1 в логические значения и из них.

К счастью, в Eloquent для этого предусмотрена система *приведения атрибутов*. Она позволяет определить, что любой из ваших столбцов должен всегда рассматриваться при чтении и записи так, как если бы они были конкретного типа данных. Варианты перечислены в табл. 5.1.

Таблица 5.1. Возможные типы столбцов приведения атрибутов

Тип	Описание
int integer	Совпадает с PHP (int)
real float double	Совпадает с PHP (float)
string	Совпадает с PHP (string)
bool boolean	Совпадает с PHP (bool)
object	Преобразуется в/из JSON как объект stdClass
array	Преобразуется в/из JSON как массив
collection	Преобразуется в/из JSON как коллекция
date datetime	Преобразуется из DATETIME БД в Carbon и обратно
timestamp	Преобразуется из TIMESTAMP БД в Carbon и обратно

Пример 5.38 показывает, как использовать приведение атрибутов в вашей модели.

Пример 5.38. Использование приведения атрибутов в модели Eloquent

```
class Contact
{
    protected $casts = [
        'vip' => 'boolean',
        'children_names' => 'array',
        'birthday' => 'date',
    ];
}
```

Мутаторы даты

Вы можете выбрать для определенных столбцов мутирование в качестве `timestamp`, добавив их в массив `dates`, как показано в примере 5.39.

Пример 5.39. Определение столбцов, которые должны мутировать в метки времени

```
class Contact
{
    protected $dates = [
        'met_at',
    ];
}
```

По умолчанию этот массив содержит `created_at` и `updated_at`, поэтому добавление записей в `dates` просто включает их в список.

5.2 Однако нет никакой разницы между добавлением столбцов в этот список и добавлением их в список `$this->casts` в качестве `timestamp`. То есть теперь данная возможность уже фактически стала ненужной, поскольку начиная с Laravel 5.2 мы можем приводить атрибуты к типу `timestamp`.

Коллекции Eloquent

Когда вы выполняете любой запрос в Eloquent, который может вернуть несколько строк, вместо массива они упаковываются в коллекцию Eloquent, которая представляет собой специализированный тип коллекции. Посмотрим на обычные коллекции и коллекции Eloquent. Рассмотрим, что делает их лучше, чем простые массивы.

Введение в базовую коллекцию

Объекты `Laravel Collection (Illuminate\Support\Collection)` немного похожи на массивы на стероидах. Методы, применяемые к объектам, подобным массивам, настолько полезны, что через некоторое время вы захотите включить их в проекты, не относящиеся к Laravel. В этом поможет пакет `Tightenco/Collect`.

Самый простой способ создать коллекцию — использовать хелпер `collect()`. Можно передать массив или применять его без аргументов, чтобы создать пустую коллекцию, а позже вставить в нее элементы. Попробуем:

```
$collection = collect([1, 2, 3]);
```

Теперь предположим, что мы хотим отфильтровать все четные числа:

```
$odds = $collection->reject(function ($item) {
    return $item % 2 === 0;
});
```

Или что, если мы хотим получить версию коллекции, в которой каждый элемент умножается на 10? Мы можем сделать это следующим образом:

```
$multiplied = $collection->map(function ($item) {  
    return $item * 10;  
});
```

Мы даже можем получить только четные числа, умножить их на 10 и вывести сумму с помощью `sum()`:

```
$sum = $collection  
->filter(function ($item) {  
    return $item % 2 == 0;  
})->map(function ($item) {  
    return $item * 10;  
})->sum();
```

Коллекции предоставляют ряд методов, которые при необходимости могут быть объединены в цепочку, для выполнения функциональных операций над вашими массивами. Они обеспечивают ту же функциональность, что и родные методы PHP, такие как `array_map()` и `array_reduce()`, но вам не нужно запоминать непредсказуемый порядок параметров в PHP, а синтаксис цепочки методов намного более читабелен.

В классе `Collection` доступно более 60 методов, включая `max()`, `whereIn()`, `flatten()` и `flip()`, и здесь недостаточно места, чтобы описать их все. Мы поговорим о методах подробнее в главе 17, или вы можете воспользоваться документацией по коллекциям Laravel (<https://laravel.com/docs/master/collections>), чтобы ознакомиться со всеми методами.



Коллекции вместо массивов

Коллекции также можно использовать в любом контексте (кроме подсказок при вводе кода), где применимы массивы. Они допускают итерацию, поэтому вы можете передавать их в `foreach`, и доступ к массиву, поэтому, если в них присвоены ключи, можно попробовать `$a = $collection['a']`.

Преимущества коллекций Eloquent

Коллекции Eloquent похожи на обычные, но с некоторыми специфическими особенностями. Если кратко, они касаются уникальных аспектов взаимодействия с коллекцией не только общих объектов, но и объектов, предназначенных для представления строк базы данных.

Например, во всех коллекциях Eloquent есть метод `modelKeys()`, который возвращает массив первичных ключей каждого экземпляра в коллекции. `find($id)` ищет экземпляр с первичным ключом `$id`.

Одна дополнительная доступная функция — это возможность определить, что любая заданная модель должна возвращать свои результаты обернутыми в определенный класс коллекции. Таким образом, если вы хотите добавить определенные методы в любую коллекцию объектов вашей модели `Order` — возможно, связанную с обобщением финансовых деталей ваших заказов, — вы можете создать собственную коллекцию `OrderCollection`, которая расширяет `Illuminate\Database\Eloquent\Collection`, а затем зарегистрировать ее в вашей модели, как показано в примере 5.40.

Пример 5.40. Пользовательские классы `Collection` для моделей Eloquent

```
...
class OrderCollection extends Collection
{
    public function sumBillableAmount()
    {
        return $this->reduce(function ($carry, $order) {
            return $carry + ($order->billable ? $order->amount : 0);
        }, 0);
    }
}
...
class Order extends Model
{
    public function newCollection(array $models = [])
    {
        return new OrderCollection($models);
    }
}
```

Теперь всякий раз, когда вы возвращаете коллекцию `Orders` (например, из `Order::all()`), она фактически будет экземпляром класса `OrderCollection`:

```
$orders = Order::all();
$billableAmount = $orders->sumBillableAmount();
```

Сериализация Eloquent

Сериализация — это то, что происходит, когда вы берете что-то сложное (массив/объект) и конвертируете в строку. В веб-контексте этой строкой часто является JSON, но она может принимать и другие формы.

Сериализация сложных записей в базе данных может быть сложной, и это одно из тех мест, где многие ORM не особо справляются. К счастью, вы получаете два мощных метода бесплатно с Eloquent: `toArray()` и `toJson()`. В коллекциях также имеются `toArray()` и `toJson()`, поэтому следующие примеры допустимы:

```
$contactArray = Contact::first()->toArray();
$contactJson = Contact::first()->toJson();
$contactsArray = Contact::all()->toArray();
$contactsJson = Contact::all()->toJson();
```

Вы также можете привести экземпляр или коллекцию Eloquent к строке (`$string = (string)$contact;`), но как модели, так и коллекции просто запустят `toJson()` и вернут результат.

Возврат моделей напрямую из методов маршрута

Маршрутизатор Laravel в конечном итоге преобразует все возвращаемые маршруты в строку, но есть хитрый прием. Если вы вернете результат вызова Eloquent в контроллере, он будет автоматически приведен к строке и, следовательно, возвращен как JSON. Это означает, что маршрут, возвращающий JSON, может быть так же прост, как и любой из маршрутов примера 5.41.

Пример 5.41. Возврат JSON из маршрутов напрямую

```
// routes/web.php
Route::get('api/contacts', function () {
    return Contact::all();
});
Route::get('api/contacts/{id}', function ($id) {
    return Contact::findOrFail($id);
});
```

Соккрытие атрибутов от JSON

Возврат JSON часто используется в API, и в этом контексте часто требуется скрывать определенные атрибуты. В силу этого Eloquent позволяет легко скрывать любые атрибуты при каждом приведении к JSON.

Вы можете занести в черный список атрибуты, чтобы скрыть их:

```
class Contact extends Model
{
    public $hidden = ['password', 'remember_token'];
```

Или занести атрибуты в белый список, чтобы показать только их:

```
class Contact extends Model
{
    public $visible = ['name', 'email', 'status'];
```

Это также работает для связей:

```
class User extends Model
{
    public $hidden = ['contacts'];

    public function contacts()
    {
        return $this->hasMany(Contact::class);
    }
}
```



Загрузка контента связей

По умолчанию контенты связей не загружаются при получении записи в базе данных, поэтому не важно, скрываете вы их или нет. Но можно получить запись со связанными с ней элементами, и в этом контексте они не будут включены в сериализованную копию записи, если вы решите скрыть эту связь.

Если вам интересно, вы можете получить User со всеми контактами — при условии, что вы правильно настроили связь, — с помощью следующего вызова:

```
$user = User::with('contacts')->first();
```

Можно сделать атрибут видимым только для одного вызова с помощью метода Eloquent `makeVisible()`:

```
$array = $user->makeVisible('remember_token')->toArray();
```



Добавление сгенерированного столбца в массив и вывод JSON

Если вы создали аксессор для несуществующего столбца — например, для столбца `full_name` из примера 5.35, — добавьте его в массив `$appends` в модели, чтобы добавить его в массив и вывод JSON:

```
class Contact extends Model
{
    protected $appends = ['full_name'];

    public function getFullNameAttribute()
    {
        return "{$this->first_name} {$this->last_name}";
    }
}
```

Связи в Eloquent

В модели реляционной базы данных ожидается, что у вас будут *связанные* друг с другом таблицы — откуда и название. Eloquent предоставляет простые и мощные инструменты, которые делают процесс связывания таблиц базы данных проще, чем когда-либо прежде.

Многие из наших примеров в этой главе были сосредоточены вокруг *пользователя* с множеством *контактов*, что довольно распространенная ситуация.

В ORM, подобном Eloquent, вы можете называть это связью «*один ко многим*»: у одного пользователя много контактов.

Будь это CRM, где контакт мог быть назначен многим пользователям, — это была бы связь «*многие ко многим*»: многие пользователи могут быть связаны с одним контактом, а каждый пользователь может быть связан со многими

контактами. У пользователя *есть много* контактов, и он *относится ко многим* контактам.

Если у каждого контакта может быть много телефонных номеров и пользователю требуется база данных каждого телефонного номера для его CRM, то у пользователя *есть много* телефонных номеров *через* контакты. То есть у пользователя *много* контактов, а у контакта *много* номеров, так что контакт является своего рода промежуточным звеном.

А что, если у каждого контакта есть адрес, но вы заинтересованы только в отслеживании одного адреса? У вас могут быть все поля адреса в `Contact`, но вы также можете создать модель `Address`, подразумевая, что у контакта *есть один* адрес.

Наконец, что, если вы хотите отмечать звездочкой (любимые) контакты, а также события? Это будут полиморфные связи, когда у пользователя много звездочек, но одни могут быть контактами, а другие — событиями.

Итак, подробнее разберем, как определить эти связи и получить к ним доступ.

Один к одному

Начнем с простого: у `Contact` есть один `PhoneNumber`. Эта связь определена в примере 5.42.

Пример 5.42. Определение связи «один к одному»

```
class Contact extends Model
{
    public function phoneNumber()
    {
        return $this->hasOne(PhoneNumber::class);
    }
}
```

Можно сказать, что методы, определяющие связи, находятся в самой модели Eloquent (`$this->hasOne()`) и принимают, по крайней мере в этом экземпляре, полное имя класса, к которому вы их относите.

Как это должно быть определено в вашей базе данных? Поскольку мы задали, что у `Contact` есть один `PhoneNumber`, Eloquent ожидает, что в таблице, поддерживающей класс `PhoneNumber` (вероятно, `phone_numbers`), есть столбец `contact_id`. Если вы назвали его по-другому (например, `owner_id`), нужно изменить определение:

```
return $this->hasOne(PhoneNumber::class, 'owner_id');
```

Так мы получаем доступ к `PhoneNumber` `Contact`:

```
$contact = Contact::first();
$contactPhone = $contact->phoneNumber;
```

Обратите внимание, что мы определяем метод в примере 5.42 с помощью `phoneNumber()`, но обращаемся к нему посредством `->phoneNumber`. Это магия. Вы также

можете получить к нему доступ, используя `->phone_number`. Это вернет полный экземпляр Eloquent связанной записи `PhoneNumber`.

Но что, если мы хотим получить доступ к `Contact` из `PhoneNumber`? Для этого тоже есть метод (пример 5.43).

Пример 5.43. Определение обратной связи «один к одному»

```
class PhoneNumber extends Model
{
    public function contact()
    {
        return $this->belongsTo(Contact::class);
    }
}
```

Затем мы получаем к нему доступ таким же образом:

```
$contact = $phoneNumber->contact;
```



Вставка связанных элементов

У каждого типа связей есть особенности для связи моделей, но вот суть их работы: экземпляр передается `save()` или массив экземпляров — `saveMany()`. Вы также можете передать свойства в `create()` или `createMany()`, и они создадут для вас новые экземпляры:

```
$contact = Contact::first();

$phoneNumber = new PhoneNumber;
$phoneNumber->number = 8008675309;
$contact->phoneNumbers()->save($phoneNumber);
```

// или

```
$contact->phoneNumbers()->saveMany([
    PhoneNumber::find(1),
    PhoneNumber::find(2),
]);
```

// или

```
$contact->phoneNumbers()->create([
    'number' => '+13138675309',
]);
```

// или

```
$contact->phoneNumbers()->createMany([
    ['number' => '+13138675309'],
    ['number' => '+15556060842'],
]);
```



Метод `createMany()` доступен только в Laravel 5.4 и более поздних версиях.

Один ко многим

Связь «один ко многим» наиболее распространена. Посмотрим, как определить, что у нашего `User` *много* `Contacts` (пример 5.44).

Пример 5.44. Определение связи «один ко многим»

```
class User extends Model
{
    public function contacts()
    {
        return $this->hasMany(Contact::class);
    }
}
```

Предполагается, что в соответствующей таблице модели `Contact` (вероятно, `contacts`) есть столбец `user_id`. Если это не так, переопределите его, передав правильное имя столбца в качестве второго параметра в `hasMany()`.

Мы можем получить `Contacts` для `User` следующим образом:

```
$user = User::first();
$usersContacts = $user->contacts;
```

Как и в случае с «одним к одному», мы используем имя метода связей и вызываем его, как если бы оно было свойством, а не методом. Однако так возвращается коллекция вместо экземпляра модели. И это нормальная коллекция Eloquent, поэтому мы можем ее обрабатывать:

```
$donors = $user->contacts->filter(function ($contact) {
    return $contact->status == 'donor';
});

$lifetimeValue = $contact->orders->reduce(function ($carry, $order) {
    return $carry + $order->amount;
}, 0);
```

Как и с «одним к одному», мы можем определить обратную связь (пример 5.45).

Пример 5.45. Определение обратной связи «один ко многим»

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

И так же, как в случае с «одним к одному», мы можем получить доступ к `User` из `Contact`:

```
$userName = $contact->user->name;
```



Присоединение и отсоединение связанных элементов от прикрепленного элемента

В большинстве случаев мы прикрепляем элемент, выполняя `save()` в родительском элементе и передавая связанный элемент, например `$user->contacts()->save($contact)`. Но если вы хотите сделать это в присоединенном («дочернем») элементе, то можете использовать `associate()` и `dissociate()` в методе, при этом возвращается связь `belongsToMany`:

```
$contact = Contact::first();

$contact->user()->associate(User::first());
$contact->save();

// и позже

$contact->user()->dissociate();
$contact->save();
```

Использование связей в качестве генераторов запросов. До сих пор мы брали имя метода (например, `contacts()`) и вызывали его, словно это свойство (например, `$user->contacts`). Что произойдет, если мы будем вызывать метод как метод? Вместо того чтобы обработать связь, он вернет генератор запросов с предварительно заданной областью действия.

Поэтому, если у вас есть `User 1` и вы вызываете его метод `contacts()`, генератор запросов будет с предопределенной областью действия «все контакты, у которых поле `user_id` имеет значение 1». Оттуда вы можете создать функциональный запрос.

```
$donors = $user->contacts()->where('status', 'donor')->get();
```

Выбор только записей со связанным элементом. Вы можете выбрать только записи, соответствующие определенным критериям в отношении связанных элементов, используя `has()`:

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

Можно настроить критерии дальше:

```
$postsWithManyComments = Post::has('comments', '>=', 5)->get();
```

Или вложить критерии:

```
$usersWithPhoneBooks = User::has('contacts.phoneNumbers')->get();
```

Наконец, вы можете написать пользовательские запросы в связанных элементах:

```
// Получает все контакты с номером телефона, содержащим строку "867-5309"
$jennyIGotYourNumber = Contact::whereHas('phoneNumbers', function ($query) {
    $query->where('number', 'like', '%867-5309%');
}));
```

Доступ ко многим через (hasManyThrough)

`hasManyThrough()` — действительно удобный метод для получения отношений между связями. Вспомните пример, когда у `User` есть много `Contacts`, а у каждого `Contact` есть много `PhoneNumbers`. Что, если вы хотите получить список контактов пользователя? Это связь «доступ ко многим через».

Структура предполагает, что в вашей таблице `contacts` есть `user_id` для связи контактов с пользователями, а в таблице `phone_numbers` есть `contact_id`, связанный с контактами. Затем мы определяем связи в `User`, как показано в примере 5.46.

Пример 5.46. Определение связи «доступ ко многим через»

```
class User extends Model
{
    public function phoneNumbers()
    {
        return $this->hasManyThrough(PhoneNumber::class, Contact::class);
    }
}
```

Вы можете получить доступ к этой связи с помощью `$user->phone_numbers`, настроить ключ связи в промежуточной модели (с третьим параметром `hasManyThrough()`) и ключ связи в удаленной модели (четвертый параметр).

Доступ к одному через (hasOneThrough)

`hasOneThrough()` аналогичен `hasManyThrough()`, но вместо обращения ко многим связанным элементам через промежуточные элементы вы получаете доступ только к одному связанному элементу через один промежуточный элемент.

Допустим, каждый пользователь относится к некой компании, у которой только один телефонный номер. Вам нужно получить номер телефона пользователя, узнав контакт его компании. Для этого есть `hasOneThrough()`.

Пример 5.47. Определение связи «доступ к одному через»

```
class User extends Model
{
    public function phoneNumber()
    {
        return $this->hasOneThrough(PhoneNumber::class, Company::class);
    }
}
```

Многие ко многим

Здесь начинаются сложности. Рассмотрим пример CRM, который позволяет `User` иметь много `Contacts` и каждый `Contact` может быть связан с несколькими `Users`.

Сначала мы определяем связь в `User`, как в примере 5.48.

Пример 5.48. Определение связи «многие ко многим»

```
class User extends Model
{
    public function contacts()
    {
        return $this->belongsToMany(Contact::class);
    }
}
```

В случае «многие ко многим» обратная связь выглядит точно так же (пример 5.49).

Пример 5.49. Определение обратной связи «многие ко многим»

```
class Contact extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Поскольку один `Contact` не может содержать столбец `user_id`, а у одного `User` не может быть столбца `contact_id`, связи «многие ко многим» основаны на сводной связывающей их таблице. Обычно ее именование выполняется путем совмещения двух имен отдельных таблиц, упорядоченных в алфавитном порядке, и разделения их подчеркиванием.

Поскольку мы связываем `users` и `contacts`, наша сводная таблица будет называться `contact_users` (если вы хотите задать имя таблицы, передайте его в качестве второго параметра методу `assignToMany()`). В ней обязательно должны быть два столбца: `contact_id` и `user_id`.

Как и в случае `hasMany()`, мы получаем доступ к коллекции связанных элементов, но на этот раз она с обеих сторон (пример 5.50).

Пример 5.50. Доступ к связанным элементам с обеих сторон связи «многие ко многим»

```
$user = User::first();

$user->contacts->each(function ($contact) {
    // сделать что-нибудь
});

$contact = Contact::first();

$contact->users->each(function ($user) {
    // сделать что-нибудь
});

$donors = $user->contacts()->where('status', 'donor')->get();
```

Получение данных из сводной таблицы. Уникальность связи «многие ко многим» состоит в том, что это наша первая связь со сводной таблицей. Лучше меньше данных в сводной таблице, но иногда там полезно хранить определенную информацию — например, можно хранить поле `created_at`, чтобы видеть дату создания связи.

Чтобы сохранить эти поля, нужно добавить их в конкретные связи, как в примере 5.51. Вы можете задать определенные поля, используя `withPivot()`, или добавить метки времени `created_at` и `updated_at` с помощью `withTimestamps()`.

Пример 5.51. Добавление полей в сводную запись

```
public function contacts()
{
    return $this->belongsToMany(Contact::class)
        ->withTimestamps()
        ->withPivot('status', 'preferred_greeting');
}
```

Когда вы получаете экземпляр модели через связь, у него будет свойство `pivot`, представляющее его место в сводной таблице, из которой его извлекли. Итак, можно сделать что-то вроде того, что показано в примере 5.52.

Пример 5.52. Получение данных из сводной записи связанного элемента

```
$user = User::first();

$user->contacts->each(function ($contact) {
    echo sprintf(
        'Contact associated with this user at: %s',
        $contact->pivot->created_at
    );
});
```

Если хотите, назовите `pivot` по-другому с помощью метода `as()`, как показано в примере 5.53.

Пример 5.53. Настройка имени сводного атрибута

```
// Пользовательская модель
public function groups()
{
    return $this->belongsToMany(Group::class)
        ->withTimestamps()
        ->as('membership');
}

// Использование этой связи:
User::first()->groups->each(function ($group) {
    echo sprintf(
        'User joined this group at: %s',
        $group->membership->created_at
    );
});
```

ОСОБЕННОСТИ ПРИСОЕДИНЕНИЯ И ОТСОЕДИНЕНИЯ СВЯЗАННЫХ ЭЛЕМЕНТОВ «МНОГИЕ КО МНОГИМ»

Поскольку ваша сводная таблица может иметь собственные свойства, нужна возможность установки этих свойств при присоединении связанного элемента «многие ко многим». Можно сделать это, передав массив в качестве второго параметра в `save()`:

```
$user = User::first();
$contact = Contact::first();
$user->contacts()->save($contact, ['status' => 'donor']);
```

Кроме того, вы можете использовать `attach()` и `detach()`, и вместо передачи экземпляра связанного элемента просто отправить идентификатор. Они работают так же, как `save()`, но могут принимать массив идентификаторов без необходимости переименовывать метод во что-то вроде `attachMany()`:

```
$user = User::first();
$user->contacts()->attach(1);
$user->contacts()->attach(2, ['status' => 'donor']);
$user->contacts()->attach([1, 2, 3]);
$user->contacts()->attach([
    1 => ['status' => 'donor'],
    2,
    3,
]);

$user->contacts()->detach(1);
$user->contacts()->detach([1, 2]);
$user->contacts()->detach(); // Отсоединяет все контакты
```

Если ваша цель не присоединение или отсоединение, а просто инвертирование текущего состояния связывания, используйте `toggle()`. Если данный идентификатор в данный момент присоединен, он будет отсоединен, и наоборот:

```
$user->contacts()->toggle([1, 2, 3]);
```

Вы также можете использовать `updateExistingPivot()`, чтобы изменить только сводную запись:

```
$user->contacts()->updateExistingPivot($contactId, [
    'status' => 'inactive',
]);
```

Если нужно заменить текущие связи, эффективно отсоединив все предыдущие и добавив новые, передайте массив в `sync()`:

```
$user->contacts()->sync([1, 2, 3]);
$user->contacts()->sync([
    1 => ['status' => 'donor'],
    2,
    3,
]);
```

Полиморфные связи

Помните, что наши полиморфные связи — это несколько классов Eloquent, соответствующих одной и той же связи. Мы собираемся использовать *Stars* (чтобы отметить избранное) прямо сейчас. Пользователь может пометить как *Contacts*, так и *Events*, отсюда и название «*полиморфный*»: существует единый интерфейс для объектов нескольких типов.

Нам понадобятся три таблицы и три модели: *Star*, *Contact* и *Event* (технически, по четыре каждой из них, потому что нам понадобятся *users* и *User* и т. д.). Таблицы *contacts* и *events* будут обычными, а таблица *stars* — с полями *id*, *starrable_id* и *starrable_type*. Для каждой *Star* мы будем определять тип (например, *Contact* или *Event*) и идентификатор этого типа (например, 1).

Создадим наши модели, как показано в примере 5.54.

Пример 5.54. Создание моделей для полиморфной системы отметок

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphTo();
    }
}

class Contact extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}

class Event extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}
```

Итак, как мы создаем *Star*?

```
$contact = Contact::first();
$contact->stars()->create();
```

Все просто. *Contact* теперь помечен.

Чтобы найти все *Stars* в данном *Contact*, мы вызываем метод *stars()*, как в примере 5.55.

Пример 5.55. Получение экземпляров полиморфной связи

```
$contact = Contact::first();

$contact->stars->each(function ($star) {
    // Делаем что-то
});
```

Если у нас есть экземпляр *Star*, можно получить его цель вызовом метода, использованного для определения его связи *morphTo*. Здесь это *starrable()*. Взгляните на пример 5.56.

Пример 5.56. Получение цели полиморфного экземпляра

```
$stars = Star::all();

$stars->each(function ($star) {
    var_dump($star->starrable()); // Экземпляр Contact или Event
});
```

Узнать, кто отметил этот контакт, так же просто, как добавить *user_id* к вашей таблице *stars*, а затем установить, что у *User* *много Stars*, а *Star принадлежит* одному *User* — связь «один ко многим» (пример 5.57). Таблица *stars* становится почти сводной таблицей для ваших *User*, *Contacts* и *Events*.

Пример 5.57. Расширение полиморфной системы для дифференциации по пользователям

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphsTo;
    }

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

class User extends Model
{
    public function stars()
    {
        return $this->hasMany(Star::class);
    }
}
```

Теперь можно запустить *\$star->user* или *\$user->stars*, чтобы найти список *Stars* для *User* или *User*, который сделал отметку, из *Star*. Кроме того, теперь при создании нового экземпляра *Star* вам нужно передать *User*:

```
$user = User::first();
$event = Event::first();
$event->stars()->create(['user_id' => $user->id]);
```


Полиморфная связь «многие ко многим»

Это самый сложный и наименее распространенный тип связей. Полиморфные связи «многие ко многим» похожи на обычные полиморфные связи, за исключением того, что это «многие ко многим».

Наиболее распространенный пример этого типа связей — тег. Представим, что вы хотите тегировать `Contacts` и `Events`. Каждый тег может применяться к нескольким элементам, а каждый тегированный элемент может иметь несколько тегов. Кроме того, это полиморфно: теги могут относиться к элементам нескольких разных типов. Для базы данных мы начнем с нормальной структуры полиморфных связей, но добавим сводную таблицу.

Нам понадобятся таблицы `contacts`, `events` и `tags`, организованные как обычно, с идентификатором и любыми свойствами, и новая таблица `taggables` с полями `tag_id`, `taggable_id` и `taggable_type`. Каждая запись в таблице `taggables` будет связывать тег с одним из типов тегировемого содержимого.

Теперь определим эту связь в наших моделях, как показано в примере 5.58.

Пример 5.58. Определение полиморфной связи «многие ко многим»

```
class Contact extends Model
{
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class Event extends Model
{
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class Tag extends Model
{
    public function contacts()
    {
        return $this->morphedByMany(Contact::class, 'taggable');
    }
    public function events()
    {
        return $this->morphedByMany(Event::class, 'taggable');
    }
}
```

Так можно создать свой первый тег:

```
$tag = Tag::firstOrCreate(['name' => 'likes-cheese']);  
$contact = Contact::first();  
$contact->tags()->attach($tag->id);
```

Мы получаем результаты этой связи как обычно, что показано в примере 5.59.

Пример 5.59. Доступ к связанным элементам с обеих сторон полиморфной связи «многие ко многим»

```
$contact = Contact::first();  
  
$contact->tags->each(function ($tag) {  
    // Делаем что-нибудь  
});  
  
$tag = Tag::first();  
  
$tag->contacts->each(function ($contact) {  
    // Делаем что-нибудь  
});
```

Обновление меток времени родительской записи дочерними записями

Любые модели Eloquent по умолчанию будут иметь метки времени `created_at` и `updated_at`. Eloquent автоматически устанавливает `updated_at` при каждом изменении в записи.

Когда связанный элемент имеет связь `belongsTo` или `belongsToMany` с другим элементом, полезно пометить этот другой элемент как обновленный всегда, когда обновляется связанный элемент. Например, если `PhoneNumber` обновлен, связанный с ним `Contact` также следует пометить как обновленный.

Добавьте имя метода для этой связи в свойство массива `$touches` в дочернем классе, как в примере 5.60.

Пример 5.60. Обновление родительской записи при каждом обновлении дочерней записи

```
class PhoneNumber extends Model  
{  
    protected $touches = ['contact'];  
  
    public function contact()  
    {  
        return $this->belongsTo(Contact::class);  
    }  
}
```

Безотлагательная загрузка

По умолчанию Eloquent загружает связи с помощью «отложенной загрузки» (*lazy loading*) — при первой загрузке экземпляра модели связанные с ним модели не будут загружаться. Скорее, они будут загружены только после того, как вы затронете их в модели. Они «ленивы» и не выполняют никакой работы, пока их не вызовут.

Это может стать проблемой, если вы перебираете список экземпляров модели и у каждого есть связанный элемент (или элементы), с которым вы работаете. Проблема с отложенной загрузкой в том, что она может привести к значительной загрузке базы данных (часто возникает проблема $N + 1$; если нет, просто игнорируйте это замечание в скобках). Например, каждый раз, когда выполняется цикл в примере 5.61, он запускает новый запрос к базе данных, чтобы найти номера телефонов для этого *Contact*.

Пример 5.61. Получение одного связанного элемента для каждого элемента в списке ($N + 1$)

```
$contacts = Contact::all();

foreach ($contacts as $contact) {
    foreach ($contact->phone_numbers as $phone_number) {
        echo $phone_number->number;
    }
}
```

Если вы загружаете экземпляр модели и знаете, что будете работать с его связями, можно вместо этого решить «безотлагательно загрузить» один или несколько его наборов связанных элементов:

```
$contacts = Contact::with('phoneNumbers')->get();
```

Использование метода *with()* с извлечением позволяет отобразить все элементы, относящиеся к полученным элементам. Как видно в примере, вы передаете имя метода, которым определяется связь.

Когда мы используем немедленную загрузку вместо извлечения связанных элементов по одному по мере поступления запросов (например, выбирая номера телефонов одного контакта каждый раз, когда выполняется цикл *foreach*), у нас формируется два запроса. Первый — чтобы извлечь начальные элементы (выбирая все контакты), и второй — чтобы получить все связанные с ними элементы (выбирая все номера телефонов, принадлежащие только что извлеченным контактам).

Вы можете безотлагательно загрузить несколько связей, передав ряд параметров в вызов *with()*:

```
$contacts = Contact::with('phoneNumbers', 'addresses')->get();
```

И вы можете использовать вложенную безотлагательную загрузку, чтобы немедленно загрузить отношения между связями:

```
$authors = Author::with('posts.comments')->get();
```

Ограничение безотлагательных загрузок. Если нужно немедленно загрузить связи, но не все элементы, можно передать замыкание в `with()`, чтобы точно определить, какие связанные элементы следует загрузить прямо сейчас:

```
$contacts = Contact::with(['addresses' => function ($query) {
    $query->where('mailable', true);
}])->get();
```

Ленивая безотлагательная загрузка. Звучит странно, потому что мы только что определили безотлагательную загрузку как противоположность отложенной. Но иногда вы не знаете, что нужно выполнить запрос на безотлагательную загрузку, пока не будут извлечены начальные экземпляры. В этом контексте вы все же можете сделать один запрос, чтобы найти все связанные элементы, избегая затрат $N + 1$. Мы называем это «ленивая безотлагательная загрузка»:

```
$contacts = Contact::all();
if ($showPhoneNumbers) {
    $contacts->load('phoneNumbers');
}
```

5.5 Чтобы загрузить связь, только если она еще не загружена, используйте метод `loadMissing()` (доступен с Laravel 5.5):

```
$contacts = Contact::all();

if ($showPhoneNumbers) {
    $contacts->loadMissing('phoneNumbers');
}
```

Безотлагательная загрузка только счетчика

Если вы хотите немедленно загрузить связи только для того, чтобы получить количество элементов в каждой связи, используйте `withCount()`:

```
$authors = Author::withCount('posts')->get();

// Добавляет целое число "posts_count" для каждого Author с количеством
// сообщений, связанных с этим автором
```

События Eloquent

Модели Eloquent запускают события вашего приложения каждый раз, когда происходят определенные действия, независимо от того, слушаете вы или нет. Если вы знакомы с моделью «издатель/подписчик» (pub/sub), то эта модель такая же (вы узнаете больше обо всей системе событий Laravel в главе 16).

Краткое изложение привязки слушателя при создании нового `Contact`. Мы собираемся связать его в методе `AppServiceProvider boot()`. Представим, что мы уведомляем сторонний сервис каждый раз, когда создаем новый `Contact`.

Пример 5.62. Привязка слушателя к событию Eloquent

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        $thirdPartyService = new SomeThirdPartyService;

        Contact::creating(function ($contact) use ($thirdPartyService) {
            try {
                $thirdPartyService->addContact($contact);
            } catch (Exception $e) {
                Log::error('Failed adding contact to ThirdPartyService; canceled.');
```

Можно отметить несколько особенностей в примере 5.62. Во-первых, мы используем `Modelname::eventName()` в качестве метода и передаем ему замыкание. Оно получает доступ к экземпляру модели, над которым выполняется операция. Во-вторых, нам нужно определить этого слушателя где-то в сервис-провайдере. В-третьих, если мы вернем `false`, операция отменится и `save()` или `update()` тоже будут отменены.

События, которые выдает каждая модель Eloquent:

- ☐ `creating;`
- ☐ `created;`
- ☐ `updating;`
- ☐ `updated;`
- ☐ `saving;`
- ☐ `saved;`
- ☐ `deleting;`
- ☐ `deleted;`
- ☐ `restoring;`
- ☐ `restored;`
- ☐ `retrieved.`

Большинство из них должны быть довольно понятными, за исключением `restoring` и `restored`, которые выдаются при восстановлении мягко удаленной строки. Кроме того, `saving` запускается как для `creating` и `updating`, а `saved` — для `created` и `updated`.

5.5 `retrieved` (доступно в Laravel 5.5 и более поздних версиях) выдается, когда существующая модель получена из базы данных.

Тестирование

Фреймворк тестирования приложений Laravel позволяет легко проверять вашу базу данных — не путем написания модульных тестов для Eloquent, а испытанием всего вашего приложения.

Возьмем такой сценарий. Вы хотите провести тестирование и убедиться, что на определенной странице отображается один контакт, а не другой. Часть этой логики связана с взаимодействием между URL-адресом, контроллером и базой данных, поэтому лучший способ проверить это — тест приложения. Возможно, вы думаете о макетировании вызовов Eloquent и при этом стараетесь избежать влияния системы на БД. *Не надо*. Используйте пример 5.63.

Пример 5.63. Тестирование взаимодействия с базой данных с помощью простых тестов приложения

```
public function test_active_page_shows_active_and_not_inactive_contacts()
{
    $activeContact = factory(Contact::class)->create();
    $inactiveContact = factory(Contact::class)->states('inactive')->create();

    $this->get('active-contacts')
        ->assertSee($activeContact->name)
        ->assertDontSee($inactiveContact->name);
}
```

Как видите, фабрики моделей и функции тестирования приложений Laravel отлично подходят для проверки вызовов базы данных.

Кроме того, вы можете поискать определенную запись непосредственно в БД, как в примере 5.64.

Пример 5.64. Использование `assertDatabaseHas()` для проверки определенных записей в базе данных

```
public function test_contact_creation_works()
{
    $this->post('contacts', [
        'email' => 'jim@bo.com'
    ]);
}
```

```

$this->assertDatabaseHas('contacts', [
    'email' => 'jim@bo.com'
]);
}

```

Eloquent и фреймворк баз данных Laravel тщательно протестированы. *Вам не нужно проверять их.* Или макетировать. Если вы действительно хотите избежать попадания в базу данных, используйте репозиторий, а затем верните несохраненные экземпляры ваших моделей Eloquent. Но самое важное: проверяйте логику того, как ваше приложение применяет вашу БД.

Если у вас есть пользовательские аксессоры, мутаторы, области действия или что-то еще, вы также можете испытывать их напрямую, как в примере 5.65.

Пример 5.65. Тестирование аксессоров, мутаторов и областей действия

```

public function test_full_name_accessor_works()
{
    $contact = factory(Contact::class)->make([
        'first_name' => 'Alphonse',
        'last_name' => 'Cumberbund'
    ]);

    $this->assertEquals('Alphonse Cumberbund', $contact->fullName);
}

public function test_vip_scope_filters_out_non_vips()
{
    $vip = factory(Contact::class)->states('vip')->create();
    $nonVip = factory(Contact::class)->create();

    $vips = Contact::vips()->get();

    $this->assertTrue($vips->contains('id', $vip->id));
    $this->assertFalse($vips->contains('id', $nonVip->id));
}

```

Просто старайтесь не использовать в тестах сложные «цепочки Деметры» для проверки в отношении того, был ли вызван определенный текущий стек в некотором шаблоне базы данных. Если ваше тестирование становится избыточным и сложным на уровне БД, то это потому, что вы позволяете предвзятым представлениям втягивать вас в излишне сложные системы. Будьте проще.



Различные названия для методов тестирования до Laravel 5.4

В проектах, работающих с версиями Laravel до 5.4, `assertDatabaseHas()` должен быть заменен на `seeInDatabase()`, `get()` — на `visit()`, `assertSee()` — на `see()`, а `assertDontSee()` — на `dontSee()`.

Резюме

Laravel поставляется с набором впечатляющих инструментов для работы с базами данных, включая миграции, сидеры, элегантный генератор запросов и Eloquent, мощный ActiveRecord ORM. Инструменты БД Laravel вообще не требуют от вас использования Eloquent — можно обращаться к базе данных и удобно манипулировать ею, не прибегая к написанию SQL-кода напрямую. Но добавить ORM, будь то Eloquent, Doctrine или что-то еще, легко — тогда будет удобно работать с основными инструментами базы данных Laravel.

Eloquent следует шаблону Active Record, который упрощает определение класса объектов, поддерживаемых БД, включая таблицу, в которой они хранятся, форму их столбцов, методов доступа и мутаторов. Eloquent может обрабатывать любые обычные действия SQL, а также сложные отношения, вплоть до полиморфных связей «многие ко многим».

Laravel также имеет надежную систему для тестирования баз данных, включая фабрики моделей.

6

Компоненты для клиентской части

Laravel — это прежде всего PHP-фреймворк, но в нем есть ряд компонентов, ориентированных на генерацию кода клиентской части (фронтенда). Некоторые из них, такие как пагинация и пакеты сообщений, являются хелперами (helper) PHP, предназначенными для клиентской части. Laravel также предоставляет систему сборки на основе Webpack под названием Mix и некоторые соглашения относительно ресурсов, не относящихся к PHP.



Инструменты сборки Laravel до и после Laravel 5.4

5.4 До Laravel 5.4 инструмент для создания клиентской части Laravel назывался Elixir и был основан на Gulp. В 5.4 и более поздних версиях новый инструмент сборки называется Mix, который основан на Webpack.

Поскольку Mix является ядром не-PHP-компонентов клиентской части, начнем с него.

Laravel Mix

Mix — это инструмент для сборки, который предоставляет простой пользовательский интерфейс и ряд соглашений поверх Webpack (<https://webpack.js.org/>). Основная ценность Mix — упрощение наиболее распространенных задач Webpack по сборке и компиляции с помощью более чистого API и ряда соглашений об именах и структуре приложений.

По своей сути Mix — это лишь приспособление в вашем наборе инструментов Webpack. «Файл Mix», который вы будете использовать для настройки своих конфигураций, — это просто файл конфигурации Webpack, который находится в корне вашего проекта и называется `webpack.mix.js`. Тем не менее нужная конфигурация намного проще, чем большинство настроек Webpack, и понадобится намного меньше работы, чтобы наиболее распространенные задачи компиляции ресурсов могли выполняться.

КРАТКОЕ ВВЕДЕНИЕ В WEBPACK

Webpack — это инструмент JavaScript, предназначенный для компиляции статических ресурсов. Команда Webpack описывает свою цель как объединение модулей с зависимостями и создание статических ресурсов.

Webpack похож на Gulp или Grunt: эти инструменты часто используются для обработки и объединения зависимостей для сайтов. Обычно это включает в себя запуск препроцессора CSS, такого как Sass, Less или PostCSS, копирование файлов, а также объединение и минимизацию JavaScript.

В отличие от других Webpack *особенно* сконцентрирован на объединении модулей с зависимостями и создании статических ресурсов в результате. Gulp и Grunt — это исполнители задач. Подобно Make и Rake до них, инструменты могут использоваться для автоматизации любых действий, которые повторяются и их можно запрограммировать. Все они *могут* использоваться для объединения ресурсов, но это не основная задача. В результате они могут быть ограничены в некоторых более сложных потребностях при объединении ресурсов — например, в определении того, какие из сгенерированных средств не будут применяться, чтобы убрать их из окончательного вывода.

Рассмотрим распространенный пример: запуск Sass для предварительной обработки ваших стилей CSS. В обычной среде Webpack это может выглядеть как пример 6.1.

Пример 6.1. Компиляция файла Sass в Webpack до Mix

```
var path = require('path');
var MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  entry: './src/sass/app.scss',
  module: {
    rules: [
      {
        test: /\.s[ac]ss$/,
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader",
          "sass-loader"
        ]
      }
    ]
  },
  plugins: [
    new MiniCssExtractPlugin({
      path: path.resolve(__dirname, './dist'),
      filename: 'app.css'
    })
  ]
}
```

Я видел и хуже. Нет невообразимого количества свойств конфигурации, и относительно понятно, что происходит. Но скопированный из проекта в проект код, а не написанный самостоятельно и с комфортом или в который можно внести существенные изменения. Такая работа может стать запутанной и повторяющейся.

Попробуем выполнить эту же задачу в Mix (пример 6.2).

Пример 6.2. Компиляция файла Sass в Mix

```
let mix = require('laravel-mix');

mix.sass('resources/sass/app.scss', 'public/css');
```

Все. Этот код не только бесконечно проще, но еще и охватывает просмотр файлов, синхронизацию с браузером, уведомления, предписанные структуры папок, автоматическое исправление, обработку URL и многое другое.

Структура каталога Mix

Во многом простота Mix обеспечивается предполагаемой структурой каталогов. Зачем выбирать для каждого нового приложения место, где будут исходные и скомпилированные ресурсы? Просто придерживайтесь соглашений Mix, и вам больше не придется думать об этом.

Каждое новое приложение Laravel имеет папку *ресурсов*, где, как ожидает Mix, будут располагаться ваши ресурсы клиентской части. Ваш Sass будет в `resources/sass`, Less — в `resources/less`, исходный CSS — в `resources/css`, а JavaScript будет находиться в `resources/js`. Они экспортируются в `public/css` и `public/js`.



Подкаталог ресурсов до Laravel 5.7

В версиях Laravel до 5.7 каталоги `sass`, `less` и `js` были вложены в каталог `resources/assets`, а не непосредственно в каталог `resources`.

Запуск Mix

Поскольку Mix работает в Webpack, необходимо установить несколько инструментов перед использованием.

- ❑ Сначала нужно установить Node.js. Посетите сайт Node по адресу <http://nodejs.org/>, чтобы узнать, как заставить Node.js работать.
- ❑ Как только Node (и NPM с ним) будет единожды установлен, вам не придется повторять это для каждого проекта. Теперь вы готовы установить зависимости своего проекта.

- ❑ Откройте корневой каталог проекта в своем терминале и запустите `npm install`, чтобы установить необходимые пакеты (Laravel поставляется с файлом `package.json`, подготовленным для Mix, чтобы управлять NPM).

Теперь все настроено! Вы можете запустить `npm run dev`, чтобы Webpack/Mix сработал единожды, `npm run watch`, чтобы прослушивать соответствующие изменения файлов и запускаться в ответ, или `npm run prod`, чтобы запустить Mix один раз с производственными настройками (такими как минимизация вывода). Можно запустить `npm run watch-poll`, если `npm run watch` не работает в вашей среде, или `npm run hot` для горячей замены модуля (см. об HMR в следующем разделе).

Что предоставляет Mix

Я уже упоминал, что Mix может предварительно обработать ваш CSS с помощью Sass, Less и/или PostCSS. Он также может объединять файлы любого типа, минимизировать их, переименовывать и копировать, а также копировать целые каталоги.

Кроме того, Mix может обрабатывать все разновидности современного JavaScript и обеспечивать автоматическое исправление, конкатенацию и минимизацию, в частности, как часть стека сборки JavaScript. Это облегчает настройку Browsersync, HMR и управления версиями, и есть плагины, доступные для многих других распространенных сценариев сборки.

Документация Mix (<http://bit.ly/2OqiYL>) содержит сведения обо всех этих возможностях и многом другом, но мы в следующих разделах сосредоточимся на нескольких конкретных случаях использования.

Карты кода

Карты кода (source map) работают с любым препроцессором, чтобы сообщить веб-инспектору вашего браузера, какие файлы сгенерировали скомпилированный исходный проверяемый код.

По умолчанию Mix не будет генерировать карты кода для ваших файлов. Но вы можете включить эту опцию, добавив в цепочку метод `sourceMaps()` после других вызовов Mix, как показано в примере 6.3.

Пример 6.3. Включение карт кода в Mix

```
let mix = require('laravel-mix');

mix.js('resources/js/app.js', 'public/js')
    .sourceMaps();
```

Настроив Mix таким образом, вы увидите, что карты кода отображаются в виде файла `.{filename}.map` рядом с каждым сгенерированным файлом.

Если вы используете инструменты разработки вашего браузера для проверки конкретного правила CSS или действия JavaScript без карт кода, то увидите лишь большую беспорядочную массу скомпилированного кода. С картами кода ваш браузер может точно определить строку исходного файла, будь то Sass, JavaScript или что-то еще, что сгенерировало проверяемое правило.

Препроцессоры и постпроцессоры

Мы уже рассмотрели Sass и Less, но Mix также поддерживает Stylus (пример 6.4). Вы можете связать PostCSS с любыми другими вызовами стиля (пример 6.5).

Пример 6.4. Предварительная обработка CSS с помощью Stylus

```
mix.stylus('resources/stylus/app.styl', 'public/css');
```

Пример 6.5. Предварительная обработка CSS с помощью PostCSS

```
mix.sass('resources/sass/app.scss', 'public/css')
  .options({
    postCss: [
      require('postcss-css-variables')()
    ]
  });
```

CSS без препроцессора

Для работы с препроцессором есть команда — она соберет все ваши CSS-файлы, объединит и выложит в каталог `public/css`, как если бы они прошли через препроцессор. Есть несколько вариантов (пример 6.6).

Пример 6.6. Объединение таблиц стилей с помощью Mix

```
// Объединяет все файлы из resources/css
mix.styles('resources/css', 'public/css/all.css');
```

```
// Объединяет файлы из resources/css
mix.styles([
  'resources/css/normalize.css',
  'resources/css/app.css'
], 'public/css/all.css');
```

Объединение JavaScript

Доступные для работы с обычными файлами JavaScript опции очень похожи на доступные опции обычных файлов CSS. Посмотрите на пример 6.7.

Пример 6.7. Объединение файлов JavaScript с помощью Mix

```
let mix = require('laravel-mix');

// Объединяет все файлы из resources/js
mix.scripts('resources/js', 'public/js/all.js');
```

```
// Объединяет файлы из resources/js
mix.scripts([
  'resources/js/normalize.js',
  'resources/js/app.js'
], 'public/js/all.js');
```

Обработка JavaScript

Mix упрощает использование Webpack для обработки JavaScript — например, чтобы скомпилировать код ES6 в простой JavaScript (пример 6.8).

Пример 6.8. Обработка файлов JavaScript в Mix с Webpack

```
let mix = require('laravel-mix');

mix.js('resources/js/app.js', 'public/js');
```

Эти сценарии ищут указанное имя файла в `resources/js` и выводят результат в `public/js/app.js`.

Вы можете использовать более сложные аспекты набора функций Webpack, создав файл `webpack.config.js` в корневом каталоге проекта.

Копирование файлов или каталогов

Методы `copy()` или `copyDirectory()` нужны для перемещения одного файла или целого каталога:

```
mix.copy('node_modules/pkgname/dist/style.css', 'public/css/pkgname.css');
mix.copyDirectory('source/images', 'public/images');
```

Управление версиями

Большинство советов из книги Стива Соудерса «Еще более быстрые веб-сайты» (<http://shop.oreilly.com/product/9780596522315.do>) нашли свое отражение в нашей повседневной практике разработки. Мы перемещаем сценарии вниз, сокращаем количество HTTP-запросов и делаем многое другое, зачастую даже не осознавая, откуда появились эти идеи.

Однако один из советов Стива пока не нашел должного распространения, и это совет о том, чтобы как можно дольше сохранять в кэше такие ресурсы, как сценарии, стили и изображения. Это означает, что к вашему серверу будет меньше запросов на получение последней версии ваших ресурсов. Но также пользователи, скорее всего, получат кэшированную версию ваших ресурсов, что сделает их устаревшими и, следовательно, предрасположенными к скорой поломке.

Решение этой проблемы — *управление версиями*. Добавляйте уникальный хеш к имени файла каждого ресурса *всегда, когда запускаете свой сценарий сборки*.

Тогда этот уникальный файл будет кэшироваться бесконечно — или по крайней мере до следующей сборки.

В чем проблема? Во-первых, нужно получить уникальные хеши, сгенерированные и добавленные к вашим именам файлов. Требуется также обновлять свои представления при каждой сборке, чтобы ссылаться на новые имена файлов.

Mix справится с этим, и это невероятно легко. Есть два компонента: задача управления версиями в Mix и хелпер PHP `mix()`. Вы можете создать версию ваших ресурсов, запустив `mix.version()`, как показано в примере 6.9.

Пример 6.9. `mix.version`

```
let mix = require('laravel-mix');

mix.sass('resources/sass/app.scss', 'public/css')
    .version();
```

Версия сгенерированного файла ничем не отличается — он просто называется `app.css` и находится в `public/css`.



Управление версиями ресурсов с использованием параметров запроса

Способ управления версиями в Laravel немного отличается от традиционного: управление версиями добавляется с помощью параметра запроса, а не путем изменения имен файлов. Все по-прежнему работает так же, потому что браузеры воспринимают файл как новый, но при этом поддерживается несколько крайних случаев для кэшей и балансировщиков нагрузки.

Используйте хелпер PHP `mix()` в своих представлениях, чтобы обратиться к этому файлу, как в примере 6.10.

Пример 6.10. Использование функции `mix()` в представлениях

```
<link rel="stylesheet" href="{{ mix("css/app.css") }}">
```

// Результатом будет что-то вроде этого:

```
<link rel="stylesheet" href="/css/app.css?id=5ee7141a759a5fb7377a">
```

КАК РАБОТАЕТ УПРАВЛЕНИЕ ВЕРСИЯМИ MIX

Mix генерирует файл с именем `public/mix-manifest.json`. Здесь хранится информация, необходимая хелперу `mix()` для нахождения сгенерированного файла. Пример `mix-manifest.json`:

```
{
  "/css/app.css": "/css/app.css?id=4151cf6261b95f07227e"
}
```

Vue и React

Mix может обрабатывать компоненты как Vue (с однофайловыми компонентами), так и React. Стандартный вызов `Mix.js()` обрабатывает Vue, и вы можете поменять его на вызов `react()`, если хотите создать компоненты React:

```
mix.react('resources/js/app.js', 'public/js');
```

Если вы посмотрите на образец `Laravel app.js` по умолчанию и импортируемые им компоненты (пример 6.11), то увидите, что не нужно делать ничего особенного для работы с компонентами Vue. Используйте вызов `mix.js()` в вашем `app.js`.

Пример 6.11. Конфигурация в `app.js` для работы с Vue

```
window.Vue = require('vue');
```

```
Vue.component('example-component', require('./components/ExampleComponent.vue'));
```

```
const app = new Vue({  
  el: '#app'  
});
```

Если вы переключитесь на `react()`, то нужно запустить в файле для вашего первого компонента следующее:

```
require('./components/Example');
```

Обе предустановки также содержат `Axios`, `Lodash` и `Popper.js`, поэтому не нужно тратить время на настройку экосистем Vue или React.

Горячая замена модуля

5.4 Если вы пишете отдельные компоненты с помощью Vue/React, то, вероятно, привыкли обновлять страницу каждый раз, когда ваш инструмент сборки перекомпилирует компоненты, или при использовании чего-то вроде Mix полагаться на `Browsersync` для перезагрузки контента.

Это прекрасно, но если вы работаете с одностраничными приложениями (SPA), то вы вернулись к первоначальному состоянию приложения. Это обновление стирает любое состояние, которое вы создали при использовании приложения.

Горячая замена модуля (HMR, иногда называемая *горячей перезагрузкой*) решает проблему. Эту функцию не всегда легко установить, но в Mix она включена изначально. HMR работает так, словно вы сказали `Browsersync` не перезагружать весь скомпилированный файл, а перезагружать только измененные фрагменты кода. Значит, вы можете получить в своем браузере обновленный код, но при этом сохранить созданное состояние, так как тестируете SPA в нужном месте.

Чтобы использовать HMR, нужно запустить `npm run hot` вместо `npm run watch`. Для правильной работы все ваши блоки `<script>` должны получать правильные

версии ваших файлов JavaScript. По сути, Mix загружает небольшой сервер Node по адресу `localhost:8080`, поэтому, если ваш тег `<script>` указывает на другую версию сценария, HMR не будет работать.

Самый простой способ добиться этого — использовать хелпер `mix()` для ссылки на ваши сценарии. Он будет поддерживать добавление в начало `localhost:8080` в режиме HMR или вашего домена, если вы находитесь в обычном режиме разработки. Вот как это выглядит:

```
<body>
  <div id="app"></div>

  <script src="{ { mix('js/app.js') } }"></script>
</body>
```

Если вы разрабатываете свои приложения для соединения по HTTPS — например, если запускаете `valet secure` — все ресурсы также должны обслуживаться через HTTPS-соединение. Это сложнее, поэтому лучше обратиться к документации HMR по адресу <http://bit.ly/2U2xvGb>.

Извлечение кода вендора

Наиболее распространенный шаблон связывания клиентской части, который также поддерживает Mix, в итоге генерирует по одному файлу CSS и JavaScript, включающих в себя программный код для вашего проекта и код для всех его зависимостей.

Однако это значит, что для обновления файла вендора требуется его полная перестройка и повторное кэширование, что может привести к нежелательному увеличению времени загрузки.

Mix позволяет легко извлечь весь JavaScript из зависимостей вашего приложения в отдельный файл `vendor.js`. Передайте список имен библиотек вендора в `extract()`, идущий в цепочке методов после вызова `js()`. В примере 6.12 показано, как это выглядит.

Пример 6.12. Извлечение библиотеки вендора в отдельный файл

```
mix.js('resources/js/app.js', 'public/js')
  .extract(['vue'])
```

Это выводит ваш существующий `app.js`, а затем два новых файла: `manifest.js`, который инструктирует браузер, как загружать зависимости и код приложения, и `vendor.js` со специфичным для вендора кодом.

Важно загрузить эти файлы в правильном порядке в код клиентской части — сначала `manifest.js`, затем `vendor.js` и, наконец, `app.js`:

```
<script src="{ { mix('js/manifest.js') } }"></script>
<script src="{ { mix('js/vendor.js') } }"></script>
<script src="{ { mix('js/app.js') } }"></script>
```



Извлечение всех зависимостей с помощью `extract()` в Mix 4.0+

Если ваш проект использует Laravel Mix 4.0 или выше, можете вызвать метод `extract()` без аргументов. Это позволит извлечь весь список зависимостей для вашего приложения.

Переменные среды в Mix

Как показано в примере 6.13, если префикс переменной среды (в вашем файле `.env`) начинается с `MIX_`, она станет доступной в скомпилированных файлах Mix и будет называться в соответствии с конвенцией об именах, то есть `process.env.ENV_VAR_NAME`.

Пример 6.13. Использование переменных `.env` в компиляции Mix JavaScript

```
# В вашем файле .env
MIX_BUGSNAG_KEY=1j12389g08bq1234
MIX_APP_NAME="Your Best App Now"

// В файлах, скомпилированных Mix
process.env.MIX_BUGSNAG_KEY

// Например, этот код:
console.log("Welcome to " + process.env.MIX_APP_NAME);

// Будет скомпилирован в такой:
console.log("Welcome to " + "Your Best App Now");
```

Можно получить доступ к этим переменным в файлах конфигурации вашего Webpack с помощью пакета Node `dotenv`, как показано в примере 6.14.

Пример 6.14. Использование переменных `.env` в файлах конфигурации Webpack

```
// webpack.mix.js
let mix = require('laravel-mix');
require('dotenv').config();

let isProduction = process.env.MIX_ENV === "production";
```

Предустановки клиентской части и генерация кода аутентификации

Как полнофункциональный фреймворк, Laravel имеет больше возможностей для связи с инструментами клиентской части, чем рядовой бэкенд-фреймворк. Изначально поставляется целая система сборки, о которой мы уже рассказывали, но она также выполняет сборку и содержит компоненты для Vue, включая Bootstrap, Axios и Lodash.

Предустановки клиентской части

Вы можете получить представление о фронтенд-инструментах, которые поставляются вместе с каждой новой установкой Laravel, взглянув на `package.json`, `webpack.mix.js` (или `gulpfile.js` в более старых версиях), а также на представления, файлы JavaScript и CSS в каталоге `resources`. Этот набор по умолчанию называется `Vue preset`, с ним поставляется каждый новый проект Laravel.

5.5 Но что, если вы предпочитаете работать в React? Вдруг вы хотите использовать Bootstrap, но не весь этот JavaScript? А если желаете просто вырезать некоторые куски? Войдите в *предварительные настройки* клиентской части в Laravel 5.5: это предварительно подготовленные сценарии, которые изменяют или удаляют часть/все предварительные настройки, загруженные с Vue и Bootstrap. Вы можете использовать предустановки, которые предоставляются «из коробки», или применить сторонние предустановки из GitHub.

Чтобы использовать встроенный набор предварительных установок, запустите `php artisan present preset_name`:

```
php artisan preset react
php artisan preset bootstrap
php artisan preset none
```

Существует также предустановка `vue`, которая применяется к каждому новому приложению при новой установке.

Наборы предустановок клиентской части от сторонних разработчиков

Если вы хотите создать собственный набор предварительных установок или использовать сделанный другим участником сообщества, поможет система наборов предустановок клиентской части. Существует ресурс GitHub (<http://bit.ly/2OraXt6>) для облегчения поиска отличных сторонних наборов предустановок клиентской части, которые легко загрузить. В большинстве случаев нужно выполнить следующие шаги.

- ❑ Установите пакет (например, `composer require laravel-frontend-presets/tailwindcss`).
- ❑ Загрузите набор предварительных установок (скажем, `php artisan preset tailwindcss`).
- ❑ Как и для встроенных наборов, запустите `npm install` и `npm run dev`.

Для создания собственного набора предустановок в той же организации есть репозиторий каркасов (<http://bit.ly/2U4ZLrH>), которые можно использовать для упрощения процесса.

Генерация кода аутентификации

У Laravel есть ряд маршрутов и представлений, называемых *аутентификационным шаблоном*, которые являются предварительными настройками клиентской части. Если вы запустите `php artisan make:auth`, то получите страницу входа, регистрации, новый основной шаблон для представления вашего приложения, маршруты для обслуживания этих страниц и многое другое. Обратитесь к главе 9 за более подробной информацией.

Разбивка на страницы

Разбивка на страницы (пагинация, pagination) все еще может быть чрезвычайно сложной для реализации. К счастью, Laravel имеет встроенную концепцию разбивки на страницы, а также по умолчанию есть доступ к результатам Eloquent и маршрутизатору.

Разбивка на страницы результатов из базы данных

Чаще всего пагинация используется при отображении результатов запроса к базе данных, когда на одной странице получается слишком много результатов. Eloquent и генератор запросов одновременно считывают параметр запроса `page` из запроса текущей страницы и используют его в методе `paginate()` для любых наборов результатов. Единственный параметр, который нужно передать `paginate()`, — это количество результатов, которые вы хотите получить на странице. В примере 6.15 показано, как это работает.

Пример 6.15. Разбивка на страницы ответа от генератора запросов

```
// PostsController
public function index()
{
    return view('posts.index', ['posts' => DB::table('posts')->paginate(20)]);
}
```

В примере 6.15 указано, что этот маршрут должен возвращать 20 сообщений на страницу. Так определить, на какой «странице» результатов находится текущий пользователь, основываясь на параметре запроса `page` URL, если он есть. У всех моделей Eloquent один и тот же метод `paginate()`.

Когда вы отображаете результаты в своем представлении, в вашей коллекции появляется метод `links()` (`render()` для Laravel 5.1), который будет выводить элементы управления пагинацией с именами классов из библиотеки компонентов Bootstrap, назначенными им по умолчанию (пример 6.16).

Пример 6.16. Отображение ссылок для разбивки на страницы в шаблоне

```
// posts/index.blade.php
<table>
```

```
@foreach ($posts as $post)
    <tr><td>{{ $post->title }}</td></tr>
@endforeach
</table>

{{ $posts->links() }}

// По умолчанию $posts->links() будет выводить что-то вроде этого:
<ul class="pagination">
    <li class="page-item disabled"><span>&laquo;</span></li>
    <li class="page-item active"><span>1</span></li>
    <li class="page-item">
        <a class="page-link" href="http://myapp.com/posts?page=2">2</a>
    </li>
    <li class="page-item">
        <a class="page-link" href="http://myapp.com/posts?page=3">3</a>
    </li>
    <li class="page-item">
        <a class="page-link" href="http://myapp.com/posts?page=2" rel="next">
            &raquo;
        </a>
    </li>
</ul>
```



Настройка количества ссылок на страницы в Laravel 5.7 и более поздних

Если вы хотите контролировать количество ссылок на любой стороне текущей страницы, в проектах под управлением Laravel 5.7 и более поздних версий можно легко настроить это количество с помощью метода `onEachSide()`:

```
DB::table('posts')->paginate(10)->onEachSide(3);
```

```
// Выводит:
// 5 6 7 [8] 9 10 11
```

Создание разбивщиков страниц вручную

Если вы не работаете с Eloquent или с конструктором запросов или пользуетесь сложным запросом (например, с `groupBy`), может потребоваться создание разбивщика вручную. В этом помогут классы `Illuminate\Pagination\Paginator` или `Illuminate\Pagination\LengthAwarePaginator`.

Разница между этими двумя классами в том, что `Paginator` будет предоставлять только кнопки «предыдущая» и «следующая», но не будет ссылок на каждую страницу. `LengthAwarePaginator` должен знать длину всего результата, чтобы сгенерировать ссылки для каждой отдельной страницы. Возможно, вы захотите применить `Paginator` для больших наборов результатов, чтобы ему не нужно было знать об общем количестве, так как это могло бы привести к нерациональному использованию ресурсов.

Paginator и **LengthAwarePaginator** требуют, чтобы вы вручную извлекали подмножество контента, которое хотите передать в представление. Взгляните на пример 6.17 для иллюстрации.

Пример 6.17. Создание разбивщика на страницы вручную

```
use Illuminate\Http\Request;
use Illuminate\Pagination\Paginator;

Route::get('people', function (Request $request) {
    $people = [...]; // огромный список персон

    $perPage = 15;
    $offsetPages = $request->input('page', 1) - 1;

    // Разбивщик не будет нарезать ваш массив для вас
    $people = array_slice(
        $people,
        $offsetPages * $perPage,
        $perPage
    );

    return new Paginator(
        $people,
        $perPage
    );
});
```

5.2 Синтаксис **Paginator** изменился за последние несколько версий **Laravel**, поэтому, если вы работаете с 5.1, посмотрите документацию по адресу <http://bit.ly/2U6M37I>, чтобы использовать правильный синтаксис.

Пакеты сообщений

Еще одна распространенная, но непростая функция в веб-приложениях — передача сообщений между различными компонентами приложения, когда конечная цель — поделиться ими с пользователем. Например, вашему контроллеру может потребоваться отправить сообщение проверки: «Поле `email` должно быть действительным адресом электронной почты». Однако это конкретное сообщение не обязательно передавать на уровень представления; оно фактически должно пережить перенаправление и оказаться в слое представления другой страницы. Как вы структурируете эту логику обмена сообщениями?

Illuminate\Support\MessageBag — это класс для хранения, категоризации и возврата сообщений, предназначенных конечному пользователю. Он группирует все сообщения по ключу, где ключи могут быть чем-то вроде `errors` и `messages`, и предоставляет удобные методы для получения всех своих сохраненных сообщений или только некоторых по определенному ключу и вывода этих сообщений в различных форматах.

Вы можете запустить новый экземпляр **MessageBag** вручную, как в примере 6.18. Честно говоря, вы вряд ли будете делать это вручную — всего лишь мысленный эксперимент, чтобы показать принцип.

Пример 6.18. Создание и использование пакета сообщений вручную

```
$messages = [
    'errors' => [
        'Something went wrong with edit 1!',
    ],
    'messages' => [
        'Edit 2 was successful.',
    ],
];
$messagebag = new \Illuminate\Support\MessageBag($messages);

// Проверка на ошибки; если есть – оформить и вывести
if ($messagebag->has('errors')) {
    echo '<ul id="errors">';
    foreach ($messagebag->get('errors', '<li><b>:message</b></li>') as $error) {
        echo $error;
    }
    echo '</ul>';
}
```

Пакеты сообщений тесно связаны со средствами тестирования допустимости Laravel (см. раздел «Проверка допустимости» на с. 206): когда проверка возвращает ошибки, на самом деле возвращается экземпляр **MessageBag**, который затем можно передать в свое представление или прикрепить к перенаправлению, используя `redirect('route')->withErrors($messagebag)`.

Laravel передает пустой экземпляр **MessageBag** каждому представлению, связанному с переменной `$errors`. Если вы отправили пакет сообщений с помощью `withErrors()` при перенаправлении, он присвоится этой переменной `$errors`. Каждое представление всегда предполагает, что имеет **MessageBag** `$errors`, который оно может проверить везде, где производится проверка допустимости (пример 6.19), где приведен распространенный фрагмент кода для каждой страницы.

Пример 6.19. Фрагмент пакета ошибки

```
// partials/errors.blade.php
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```



Отсутствие переменной `$errors`

Если у вас есть какие-либо маршруты, не входящие в группу промежуточного программного обеспечения web, у них не будет промежуточной сессии ПО. Им будет недоступна переменная `$errors`.

Именованные пакеты ошибок. Иногда нужно различать пакеты сообщений не только по ключу (`notices` и `errors`), но и по компонентам. Возможно, у вас есть формы входа и регистрации на одной и той же странице; как вы их различите?

Когда вы отправляете ошибки вместе с перенаправлением с помощью `withErrors()`, второй параметр — имя пакета: `redirect('dashboard')->withErrors($validator, 'login')`. Затем на панели инструментов можно использовать `$errors->login` для вызова всех методов, которые вы видели ранее: `any()`, `count()` и др.

Строковые хелперы, множественность и локализация

Разработчики склонны рассматривать блоки текста как большие участки в тегах `div`, ожидая, когда клиент добавит в них реальный контент. Мы редко занимаемся какой-либо логикой внутри этих блоков.

Но при некоторых обстоятельствах вы будете благодарны за инструменты, предоставляемые Laravel для работы со строками.

Строковые хелперы и множественность

У Laravel есть ряд хелперов для манипулирования строками. Они доступны как методы класса `Str` (например, `Str::plural()`), но для большинства еще есть глобальный хелпер (скажем, `str_plural()`).

Документация Laravel (<http://bit.ly/2HQKaFC>) их подробно описывает, но вот некоторые наиболее часто используемые строковые хелперы.

- ❑ `e()`. Краткое обозначение для `html_entities()`. Кодировывает все участки HTML в целях безопасности.
- ❑ `starts_with()`, `ends_with()`, `str_contains()`. Проверяет строку (первый параметр): начинается ли она конкретным символом, заканчивается им или содержит другую строку (второй параметр).
- ❑ `str_is()`. Проверяет, соответствует ли строка (второй параметр) определенному шаблону (первый параметр) — например, `foo*` будет соответствовать `foobar` и `foobaz`.
- ❑ `str_slug()`. Преобразует строку в часть записи типа URL с дефисами.

- ❑ `str_plural(word, count)`, `str_singular()`. Переводит слово во множественное или единственное число. Только на английском языке, например, `str_plural('dog')` возвращает `dogs`; `str_plural('dog', 1)` возвращает `dog`.
- ❑ `camel_case()`, `kebab_case()`, `snake_case()`, `studly_case()`, `title_case()`. Преобразует предоставленную строку в малые заглавные буквы.
- ❑ `str_after()`, `str_before()`, `str_limit()`. Обрезает строку и выдает подстроку. `str_after()` возвращает все после заданной строки, а `str_before()` — все до заданной строки. Оба метода принимают полную строку в качестве первого параметра и строку для вырезания в качестве второго. `str_limit()` усекает строку (первый параметр) до заданного количества символов (второй параметр).

Локализация

Локализация позволяет определить несколько языков и пометить любые строки для перевода. Вы можете установить запасной язык и даже использовать множественные варианты.

В Laravel нужно установить локаль приложения во время загрузки страницы, чтобы хелперы локализации знали, из какого набора переводов нужно брать значение. Каждая локаль обычно связана с переводом и часто будет выглядеть как «en» (для английского). Это достигается с помощью действия `App::setLocale($localeName)`, и, скорее всего, вы поместите его в сервис-провайдер. Сейчас можно поместить его в метод `boot()` `AppServiceProvider` или создать `LocaleServiceProvider`, если будете использовать больше одной локали.

НАСТРОЙКА ЛОКАЛИ ДЛЯ КАЖДОГО ЗАПРОСА

Поначалу может быть сложно понять, как Laravel «распознает» локаль пользователя или переводит. Большая часть этого зависит от вас как разработчика. Посмотрим на вероятный сценарий.

Возможно, вы обеспечили некую функциональность, чтобы пользователь мог выбрать локаль, или вы пытаетесь автоматически определить ее. В любом случае ваше приложение определит языковой стандарт, а затем вы сохраните его в параметре URL или cookie-файле сессии. Затем ваш сервис-провайдер — к примеру, что-то вроде `LocaleServiceProvider` — возьмет этот ключ и установит его как часть начальной загрузки Laravel.

Предположим, пользователь находится на странице `http://myapp.com/es/contacts`. Ваш `LocaleServiceProvider` захватит эту строку и запустит `App::setLocale('es')`. Теперь всегда при запросе перевода строки Laravel будет искать версию этой строки на испанском (`es` означает `Español`), которую вам нужно будет где-то определить.

Можно определить резервную локаль в `config/app.php`, где вы сможете найти специальный ключ для этого, `fallback_locale`. Это позволяет определить язык по

умолчанию для приложения, который Laravel будет использовать, если не найдет перевод для запрошенной локали.

Базовая локализация

Как мы вызываем переведенную строку? Существует хелпер `__($key)`, который извлекает строку для текущей локали, чтобы передать ключ, или, если она не существует, извлекает ее из локали по умолчанию. В Blade можно использовать директиву `@lang()`. Пример 6.20 демонстрирует, как работает базовый перевод. Мы будем использовать как пример ссылку «назад на панель инструментов» в верхней части страницы сведений.

Пример 6.20. Базовое использование `__()`

```
// Нормальный PHP
<?php echo __('navigation.back'); ?>

// Blade
{{ __('navigation.back') }}

// Blade directive
@lang('navigation.back')
```

Предположим, сейчас мы используем локаль `es`. Laravel будет искать файл `resources/lang/es/navigation.php`, который, как ожидается, вернет массив. Он будет искать ключ `back` в этом массиве и, если он существует, возвратит его значение. Взгляните на пример 6.21 для иллюстрации.

Пример 6.21. Использование перевода

```
// resources/lang/es/navigation.php
return [
    'back' => 'Volver al panel',
];

// routes/web.php
Route::get('/es/contacts/show/{id}', function () {
    // В этом примере локаль устанавливается вручную, а не в сервис-провайдере
    App::setLocale('es');
    return view('contacts.show');
});

// resources/views/contacts/show.blade.php
<a href="/contacts">{{ __('navigation.back') }}</a>
```



Хелпер по переводу до Laravel 5.4

В проектах, работающих с версиями Laravel до 5.4, хелпер `__()` недоступен. Вместо этого нужно использовать `trans()`, который обращается к более старой системе перевода, но работает аналогично описываемому выше, хотя не имеет доступа к системе перевода JSON.

Параметры локализации

Предыдущий пример был относительно простым. Разберемся в некоторых более сложных случаях. Что, если мы хотим задать, *к какой* панели мы возвращаемся? Посмотрите на пример 6.22.

Пример 6.22. Параметры в переводах

```
// resources/lang/en/navigation.php
return [
    'back' => 'Back to :section dashboard',
];

// resources/views/contacts/show.blade.php
{{ __('navigation.back', ['section' => 'contacts']) }}
```

Как видите, двоеточие перед словом (:section) помечает его как плейсхолдер, который будет заменен. Второй (необязательный) параметр `__()` — это массив значений для замены плейсхолдеров.

Множественность в локализации

Мы рассматривали множественность, теперь представьте, что вы определяете собственные правила для этого. Есть два способа — мы начнем с простого (пример 6.23).

Пример 6.23. Определение простого перевода с опцией множественности

```
// resources/lang/en/messages.php
return [
    'task-deletion' => 'You have deleted a task|You have successfully deleted tasks',
];

// resources/views/dashboard.blade.php
@if ($numTasksDeleted > 0)
    {{ trans_choice('messages.task-deletion', $numTasksDeleted) }}
@endif
```

У нас есть метод `trans_choice()`, который принимает количество затронутых элементов в качестве второго параметра. Из этого он будет определять, какую строку использовать.

Вы также можете применить любые определения перевода, которые совместимы с гораздо более сложным компонентом `Symfony Translation` (пример 6.24).

Пример 6.24. Использование компонента `Symfony Translation`

```
// resources/lang/es/messages.php
return [
    'task-deletion' => "{0} You didn't manage to delete any tasks." .
        "[1,4] You deleted a few tasks." .
        "[5,Inf] You deleted a whole ton of tasks.",
];
```

Сохранение строки по умолчанию в качестве ключа с помощью JSON

Общая проблема с локализацией в том, что трудно обеспечить хорошую систему для определения пространства имен ключей — например, запоминание ключа, вложенного на три/четыре уровня глубины или отсутствие уверенности в том, какой ключ следует использовать для фразы, встречающейся на сайте дважды.

Альтернатива системе пар значений «ключ/строка» — хранение ваших переводов, используя в качестве ключа строку основного языка вместо чего-то выдуманного. Вы можете указать Laravel, что работаете именно так, сохраняя файлы перевода в виде JSON в каталоге `resources/lang`, при этом имя файла соответствует локали (пример 6.25).

Пример 6.25. Использование JSON-переводов и хелпера `__()`

```
// В Blade
{{ __('View friends list') }}
```

```
// resources/lang/es.json
{
    'View friends list': 'Ver lista de amigos'
}
```

При этом используется тот факт, что хелпер по переводу `__()`, если не может найти соответствующий ключ для текущего языка, просто отобразит ключ. Если ваш ключ — строка на языке вашего приложения по умолчанию, это более разумный запасной вариант, чем, например, `widgets.friends.title`.



Переводы JSON недоступны до Laravel 5.4

Формат перевода строки JSON доступен только в Laravel 5.4 и более поздних версиях.

Тестирование

В этой главе мы сосредоточились на фронтенд-компонентах Laravel. Использование модульных тестов для них менее вероятно, но иногда они могут применяться в ваших интеграционных тестах.

Тестирование пакетов сообщений и ошибок

Существует два основных способа проверки сообщений, передаваемых в пакетах сообщений и ошибок. Во-первых, можно подготовить свои тесты приложений так, чтобы появлялось сообщение, которое будет где-то отображаться, затем перейти на эту страницу и убедиться, что сообщение видно правильно.

Во-вторых, для ошибок (это наиболее частый случай) вы можете задать утверждение, что во время сессии появились ошибки, с помощью `$this->assertSessionHasErrors($bindings = [])`. В примере 6.26 показано, как это может выглядеть.

Пример 6.26. Утверждение, что в сессии есть ошибки

```
public function test_missing_email_field_errors()
{
    $this->post('person/create', ['name' => 'Japheth']);
    $this->assertSessionHasErrors(['email']);
}
```

Чтобы пройти тест из примера 6.26, нужно добавить проверку ввода в этот маршрут. Рассмотрим это в главе 7.

Перевод и локализация

Самый простой способ проверить локализацию — тесты приложения. Установите соответствующий контекст (по URL или в сессии), посетите страницу с помощью `get()` и убедитесь, что видите соответствующий контент.

Резюме

Будучи полнофункциональным фреймворком, Laravel предоставляет инструменты и компоненты для фронтенда и бэкенда.

Mix — это слой перед Webpack, который значительно упрощает обычные задачи и параметры конфигурации. Mix позволяет легко использовать популярные препроцессоры и постпроцессоры CSS, общие этапы обработки JavaScript и многое другое.

Laravel предлагает другие служебные инструменты для клиентской части, включая инструменты для пагинации, пакетов сообщений и ошибок и локализации.

7

Получение и обработка пользовательских данных

Сайты, использующие фреймворки вроде Laravel, часто не просто обслуживают статический контент. Многие работают со сложными и смешанными источниками данных. Один из наиболее распространенных и сложных источников — это пользовательский ввод в его бесчисленных формах: пути URL, параметры запроса, данные POST и загрузки файлов.

Laravel предоставляет набор инструментов для получения, проверки, нормализации и фильтрации предоставленных пользователем данных. Мы рассмотрим их здесь.

Внедрение объекта запроса

Наиболее распространенный способ доступа к пользовательским данным в Laravel — это внедрение экземпляра объекта `Illuminate\Http\Request`. Так обеспечивается легкий доступ ко всем способам, которыми пользователи могут ввести информацию на вашем сайте: данные формы POST или JSON, запросы GET (параметры запроса) и сегменты URL.



Другие варианты доступа к данным запроса

Есть также глобальный хелпер `request()` и фасад `Request`, которые предоставляют одни и те же методы. Каждая из этих опций предоставляет весь объект `Illuminate\Request`, но пока мы рассмотрим методы, конкретно относящиеся к пользовательским данным.

Поскольку мы планируем внедрить объект `Request`, кратко рассмотрим, как получить объект `$request`, в котором мы будем вызывать все эти методы:

```
Route::post('form', function (Illuminate\Http\Request $request) {
    // $request->etc()
});
```

`$request->all()`

`$request->all()` дает вам массив, содержащий все входные данные, предоставленные пользователем, из всех источников. Допустим, по какой-то причине вы решили создать форму POST для URL-адреса с параметром запроса — например, отправив POST на адрес `http://myapp.com/signup?utm=12345`. В примере 7.1 показано, что `$request->all()` вернет запрос. `$request->all()` также содержит информацию обо всех загруженных файлах, но об этом мы поговорим позже в этой главе.

Пример 7.1. `$request->all()`

```
<!-- GET-маршрут из представления по адресу /get-route -->
<form method="post" action="/signup?utm=12345">
    @csrf
    <input type="text" name="first_name">
    <input type="submit">
</form>

// routes/web.php
Route::post('signup', function (Request $request) {
    var_dump($request->all());
});

// Выводы:
/**
 * [
 *     '_token' => 'CSRF token here',
 *     'first_name' => 'value',
 *     'utm' => 12345,
 * ]
 */
```

`$request->except()` и `$request->only()`

Вывод `$request->except()` такой же, как у `$request->all()`, но можно выбрать одно или несколько полей для исключения, например, `_token`. Вы можете передать его как строку или массив строк.

В примере 7.2 показано, как это выглядит, когда мы используем `$request->except()` в такой же форме, как в примере 7.1.

Пример 7.2. `$request->except()`

```
Route::post('post-route', function (Request $request) {
    var_dump($request->except('_token'));
});
```

```
// Выводы:
/**
 * [
 *   'firstName' => 'value',
 *   'utm' => 12345
 * ]
 */
```

`$request->only()` — инверсия `$request->except()`, как видно в примере 7.3.

Пример 7.3. `$request->only()`

```
Route::post('post-route', function (Request $request) {
    var_dump($request->only(['firstName', 'utm']));
});
```

```
// Выводы:
/**
 * [
 *   'firstName' => 'value',
 *   'utm' => 12345
 * ]
 */
```

`$request->has()`

С помощью `$request->has()` можно выяснить, доступен ли вам определенный фрагмент пользовательского ввода. В примере 7.4 показана аналитика со строчным параметром запроса `utm` из предыдущих примеров.

Пример 7.4. `$request->has()`

```
// POST маршрут в /post-route
if ($request->has('utm')) {
    // Аналитическая работа
}
```

`$request->input()`

`$request->all()`, `$request->except()` и `$request->only()` работают с полным массивом ввода, предоставленным пользователем. `$request->input()` позволяет получить значение только одного поля, как показано в примере 7.5. Обратите внимание, что второй параметр — значение по умолчанию, поэтому, если пользователь не передал значение, у вас будет разумный (и не приводящий к отказу) запасной вариант.

Пример 7.5. `$request->input()`

```
Route::post('post-route', function (Request $request) {
    $userName = $request->input('name', 'Matt');
});
```


`$request->method()` и `->isMethod()`

`$request->method()` возвращает команду HTTP для запроса, а `$request->isMethod()` проверяет, соответствует ли она указанной команде. Пример 7.6 иллюстрирует их использование.

Пример 7.6. `$request->method()` и `$request->isMethod()`

```
$method = $request->method();

if ($request->isMethod('patch')) {
    // Некие операции, если метод запроса – PATCH
}
```

Ввод массива

Laravel также предоставляет вспомогательные средства для доступа к данным из массива. Используйте нотацию «точка», чтобы указать слои погружения в структуру массива, как показано в примере 7.7.

Пример 7.7. Точечная запись для доступа к значениям массива в пользовательских данных

```
<!-- GET-маршрут представления формы в /employees/create -->
<form method="post" action="/employees/">
    @csrf
    <input type="text" name="employees[0][firstName]">
    <input type="text" name="employees[0][lastName]">
    <input type="text" name="employees[1][firstName]">
    <input type="text" name="employees[1][lastName]">
    <input type="submit">
</form>

// POST маршрут в /employees
Route::post('employees', function (Request $request) {
    $employeeZeroFirstName = $request->input('employees.0.firstName');
    $allLastNames = $request->input('employees.*.lastName');
    $employeeOne = $request->input('employees.1');
    var_dump($employeeZeroFirstName, $allLastNames, $employeeOne);
});

// Если формы заполнены следующим образом: "Jim" "Smith" "Bob" "Jones":
// $employeeZeroFirstName = 'Jim';
// $allLastNames = ['Smith', 'Jones'];
// $employeeOne = ['firstName' => 'Bob', 'lastName' => 'Jones'];
```

Ввод JSON (и `$request->json()`)

Мы рассмотрели ввод из строк запроса (GET) и отправку форм (POST). Но есть еще одна форма пользовательского ввода, которая становится все более распространенной

с появлением JavaScript SPA: запрос JSON. По сути, это просто запрос POST с телом, оформленным как JSON вместо традиционной формы POST.

Посмотрим, как это выглядит для отправки некоторого JSON в маршрут Laravel и как использовать `$request->input()` для извлечения этих данных (пример 7.8).

Пример 7.8. Получение данных из JSON с помощью `$request->input()`

```
POST /post-route HTTP/1.1
```

```
Content-Type: application/json
```

```
{
  "firstName": "Joe",
  "lastName": "Schmoe",
  "spouse": {
    "firstName": "Jill",
    "lastName": "Schmoe"
  }
}
```

```
// Post-route
```

```
Route::post('post-route', function (Request $request) {
    $firstName = $request->input('firstName');
    $spouseFirstname = $request->input('spouse.firstName');
});
```

`$request->input()` достаточно умен для извлечения пользовательских данных из GET, POST или JSON. Удивительно, что Laravel предлагает еще и `$request->json()`. Есть две причины выбрать `$request->json()`. Во-первых, для более четкого указания другим программистам, работающим над вашим проектом, откуда вы ожидаете получить данные. Во-вторых, если у POST нет правильных заголовков `application/json`, `$request->input()` не будет воспринимать содержимое как JSON, но `$request->json()` — будет.

ПРОСТРАНСТВА ИМЕН ФАСАДА, ГЛОБАЛЬНЫЙ ХЕЛПЕР REQUEST() И ВВОДЯЩИЙ \$REQUEST

Каждый раз, когда вы используете фасады внутри классов с пространствами имен (например, контроллеров), нужно добавить полный путь фасада к блоку импорта в верхней части вашего файла (например, `Illuminate\Support\Facades\Request`).

Из-за этого у некоторых фасадов есть глобальный хелпер. Если эти хелперы выполняются без параметров, они имеют тот же синтаксис, что и фасад (например, `request()->has()` — то же самое, что `Request::has()`). Им также присуще поведение по умолчанию, когда вы передаете им параметр (скажем, `request('firstName')` — просто сокращенная форма `request()->input('firstName')`).

С помощью `Request` мы рассмотрели внедрение экземпляра объекта `Request`, но вы также можете использовать фасад `Request` или глобальный хелпер `request()`. В главе 10 вы узнаете об этом больше.

Маршрутные данные

URL-адрес — это такие же пользовательские данные, как и все остальные в этой главе.

Есть два основных способа получения данных из URL-адреса: через объекты `Request` и параметры маршрута.

Из Request

У внедренных объектов `Request` (а также фасада `Request` и хелпера `request()`) есть несколько доступных методов для представления состояния URL текущей страницы, но в данный момент сосредоточимся на получении информации о сегментах URL.

Каждая группа символов после домена в URL-адресе называется *сегментом*. Например, `http://www.myapp.com/users/15/` имеет два сегмента: `users` и `15`.

У нас есть два доступных метода: `$request->segments()` возвращает массив всех сегментов, а `$request->segment($segmentId)` позволяет получить значение одного сегмента. Сегменты возвращаются на основе индекса 1, поэтому в предыдущем примере `$request->segment(1)` вернул бы `users`.

Объекты `Request`, фасад `Request` и глобальный хелпер `request()` предоставляют еще несколько методов, которые помогут нам получить данные из URL (см. главу 10).

Из параметров маршрута

Другой основной способ получения данных об URL-адресе — из встроенных в метод контроллера или замыкание параметров маршрута, которые обслуживают текущий маршрут, как показано в примере 7.9.

Пример 7.9. Получение информации URL из параметров маршрута

```
// routes/web.php
Route::get('users/{id}', function ($id) {
    // Если пользователь заходит на myapp.com/users/15/,
    // переменной $id присваивается значение 15
});
```

Чтобы узнать больше о маршрутах и привязке маршрутов, ознакомьтесь с главой 3.

Загруженные файлы

Мы узнали о разных способах взаимодействия с пользовательским вводом текста. Поговорим о загрузке файлов. Объекты `Request` предоставляют доступ к любым загруженным файлам с помощью метода `$request->file()`, который принимает на входе имя файла в качестве параметра и возвращает экземпляр `Symfony\Component\HttpFoundation\File\UploadedFile`. Рассмотрим пример. Первый — наша форма в примере 7.10.

Пример 7.10. Форма для загрузки файлов

```
<form method="post" enctype="multipart/form-data">
  @csrf
  <input type="text" name="name">
  <input type="file" name="profile_picture">
  <input type="submit">
</form>
```

Теперь посмотрим, что мы получаем при вызове `$request->all()`, как показано в примере 7.11. Заметьте, что `$request->input('profile_picture')` вернет `null`; нужно использовать вместо него `$request->file('profile_picture')`.

Пример 7.11. Вывод при отправке формы из примера 7.10

```
Route::post('form', function (Request $request) {
    var_dump($request->all());
});

// Вывод:
// [
//     "_token" => "token here",
//     "name" => "asdf",
//     "profile_picture" => UploadedFile {},
// ]

Route::post('form', function (Request $request) {
    if ($request->hasFile('profile_picture')) {
        var_dump($request->file('profile_picture'));
    }
});

// Вывод:
// UploadedFile (подробности)
```

ПРОВЕРКА ЗАГРУЗКИ ФАЙЛА

Как видно в примере 7.11, у нас есть метод `$request->hasFile()`, чтобы проверить, загрузил ли пользователь файл. Мы также можем проверить, была ли загрузка файла успешной, используя `isValid()` для самого файла:

```
if ($request->file('profile_picture')->isValid()) {
    //
}
```

Поскольку `isValid()` вызывается для самого файла, произойдет ошибка, если пользователь не загрузил файл. Чтобы проверить оба варианта, нужно узнать, существует ли файл:

```
if ($request->hasFile('profile_picture') &&
    $request->file('profile_picture')->isValid()) {
    //
}
```

Класс `UploadedFile` Symfony расширяет родной для PHP `SplFileInfo` методами, позволяющими легко проверить и манипулировать файлом. Этот список не исчерпывающий, но дает представление о возможностях:

- ☐ `guessExtension();`
- ☐ `getMimeType();`
- ☐ `store($path, $storageDisk = default disk);`
- ☐ `storeAs($path, $newName, $storageDisk = default disk);`
- ☐ `storePublicly($path, $storageDisk = default disk);`
- ☐ `storePubliclyAs($path, $newName, $storageDisk = default disk);`
- ☐ `move($directory, $newName = null);`
- ☐ `getClientOriginalName();`
- ☐ `getClientOriginalExtension();`
- ☐ `getClientMimeType();`
- ☐ `guessClientExtension();`
- ☐ `getClientSize();`
- ☐ `getError();`
- ☐ `isValid();`

5.3 Большинство методов связано с получением информации о загруженном файле, но есть один, который вы, вероятно, будете использовать чаще всех: `store()` (доступно с Laravel 5.3). Он берет загруженный с запросом файл и сохраняет его в указанном каталоге на вашем сервере. Его первым параметром является каталог назначения, а необязательным вторым параметром — диск хранения (`s3`, `local` и т. д.), на котором будет храниться файл. Общий рабочий процесс показан в примере 7.12.

Пример 7.12. Общий процесс загрузки файла

```
if ($request->hasFile('profile_picture')) {  
    $path = $request->profile_picture->store('profiles', 's3');  
    auth()->user()->profile_picture = $path;  
    auth()->user()->save();  
}
```

Если нужно указать имя файла, можно вместо `store()` использовать `storeAs()`. Первый параметр — это по-прежнему путь; вторым является имя файла, а необязательный третий параметр — используемый диск.



Правильная кодировка формы для загрузки файлов

Вы можете получить null вместо содержимого файла из запроса потому, что вы забыли установить тип кодировки в форме. Убедитесь, что добавили атрибут `enctype="multipart/form-data"` в форму:

```
<form method="post" enctype="multipart/form-data">
```

Валидация

У Laravel есть несколько способов проверить входящие данные. Мы будем рассматривать запросы формы в следующем разделе, поэтому пока у нас есть две основные возможности: проверка вручную или использование метода `validate()` объекта `Request`. Начнем с более простого и распространенного `validate()`.

Метод `validate()` объекта `Request`

У объекта `Request` есть метод `validate()`, который предоставляет удобный способ для наиболее распространенного процесса рабочей проверки. Взгляните на пример 7.13.

Пример 7.13. Основное использование проверки запроса

```
// routes/web.php
Route::get('recipes/create', 'RecipesController@create');
Route::post('recipes', 'RecipesController@store');

// app/Http/Controllers/RecipesController.php
class RecipesController extends Controller
{
    public function create()
    {
        return view('recipes.create');
    }
    public function store(Request $request)
    {
        $request->validate([
            'title' => 'required|unique:recipes|max:125',
            'body' => 'required'
        ]);

        // Рецепт действителен; продолжить, чтобы сохранить его
    }
}
```

Наши четыре строки кода проверки делают многое.

Мы явно определяем ожидаемые поля и применяем правила (здесь — разделенные символом конвейеризации, `|`) к каждому из них в отдельности.

Затем метод `validate()` проверяет входные данные из `$request` и определяет их валидность.

Если данные валидны, `validate()` завершается. Мы можем перейти к методу контроллера, сохраняя данные или делая что-то еще.

Но если данные невалидны, выдается исключение `ValidationException`. В нем содержатся инструкции для маршрутизатора, как обрабатывать это исключение. Если запрос сделан из JavaScript (или если он запрашивает JSON в качестве ответа), исключение создаст ответ JSON, содержащий ошибки валидации. Если нет,

исключение вернет перенаправление на предыдущую страницу вместе со всеми пользовательскими данными и ошибками валидации. Это прекрасно подходит для повторного заполнения формы и отображения некоторых ошибок.



Вызов метода `validate()` в контроллере до Laravel 5.5

В проектах, работающих на версиях Laravel до 5.5, этот метод валидации вызывается в контроллере с помощью `$this->validate()`, а не в запросе.

ПОДРОБНЕЕ О ПРАВИЛАХ ВАЛИДАЦИИ LARAVEL

В наших примерах мы используем синтаксис конвейеризации (как в документации): `'fieldname': 'rule|otherRule|anotherRule'`. Но также подходит синтаксис массивов: `'fieldname': ['rule', 'otherRule', 'anotherRule']`.

Кроме того, вы можете проверять вложенные свойства. Это важно, если вы используете синтаксис массива HTML, который позволяет, например, иметь несколько «пользователей» в форме HTML, каждый с соответствующим именем. Вот как можно это проверить:

```
$request->validate([
    'user.name' => 'required',
    'user.email' => 'required|email',
]);
```

Здесь недостаточно места, чтобы охватить все возможные правила валидации, но вот несколько наиболее распространенных правил и соответствующие функции.

- *Поле должно предусматривать:*
 - `required`; `required_if:anotherField,equalToThisValue`;
 - `required_unless:anotherField,equalToThisValue`.
- *Поле должно содержать определенные типы символов:* `alpha`; `alpha_dash`; `alpha_num`; `numeric`; `integer`.
- *Поле должно содержать определенные шаблоны:* `email`; `active_url`; `ip`.
- *Даты:* `after:date`; `before:date` (`date` может быть любой допустимой строкой, которую может обработать `strtotime()`).
- *Числа:* `between:min,max`; `min:num`; `max:num`; `size:num` (`size` проверяет длину для строк, значение для целых чисел, `count` для массивов или размер в КБ для файлов).
- *Размеры изображения:* `dimensions:min_width=XXX`. Может также использоваться вместе или комбинироваться с `max_width`, `min_height`, `max_height`, `width`, `height` и `ratio`.
- *Базы данных:* `exists:tableName`; `unique:tableName`. Ожидается, что поиск будет производиться в том же столбце таблицы, что и имя поля. Настройки смотрите в документации по адресу <http://bit.ly/2eMLZDI>.

Ручная валидация

Если вы не работаете в контроллере или по какой-то другой причине ранее описанный способ валидации не подходит, можно вручную создать экземпляр `Validator`, используя фасад `Validator`, и проверить успешность или неудачу, как в примере 7.14.

Пример 7.14. Ручная валидация

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

Route::post('recipes', function (Illuminate\Http\Request $request) {
    $validator = Validator::make($request->all(), [
        'title' => 'required|unique:recipes|max:125',
        'body' => 'required'
    ]);

    if ($validator->fails()) {
        return redirect('recipes/create')
            ->withErrors($validator)
            ->withInput();
    }

    // Рецепт действителен; продолжить, чтобы сохранить его
});
```

Мы создаем экземпляр валидатора, передавая ему наши входные данные в качестве первого параметра и правила валидации как второй параметр. Валидатор предоставляет метод `fails()`, предназначенный для валидации, и его можно передать в метод `withErrors()` перенаправления.

Объекты пользовательских правил

Если нужного правила валидации в Laravel нет, можно создать свое. Чтобы создать собственное правило, запустите `php artisan make:rule RuleName`, а затем отредактируйте этот файл `app/Rules/{RuleName}.php`.

Вы получите два готовых метода в своем правиле: `passes()` и `message()`. `passes()` должен принять имя атрибута в качестве первого параметра и предоставленное пользователем значение в качестве второго, а затем вернуть логическое значение, указывающее, проходит ли этот ввод данное правило валидации. Метод `message()` должен возвращать сообщение об ошибке. Можно использовать `:attribute` как плейсхолдер в вашем сообщении для имени атрибута.

Взгляните на пример 7.15.

Пример 7.15. Пример пользовательского правила

```
class WhitelistedEmailDomain implements Rule
{
    public function passes($attribute, $value)
    {
```



```
        return in_array(str_after($value, '@'), ['tighten.co']);
    }

    public function message()
    {
        return 'The :attribute field is not from a whitelisted email provider.';
    }
}
```

Для использования правила передайте экземпляр объекта правила вашему валидатору:

```
$request->validate([
    'email' => new WhitelistedEmailDomain,
]);
```



В проектах с версиями Laravel до 5.5 пользовательские правила валидации должны быть написаны с использованием `Validator::extend()`. Вы можете узнать об этом больше из документации по адресу <http://bit.ly/2Wl87J1>.

Отображение валидационных сообщений

Мы уже рассмотрели большую часть этой темы в главе 6, здесь же я вкратце напомню об отображении ошибки валидации.

Метод `validate()` для запросов (и метод `withErrors()` для перенаправлений, на которые он полагается) высвечивает любые ошибки сессии. Они становятся доступными в представлении, на которое вы перенаправляетесь, с помощью переменной `$errors`. Помните, как часть волшебства Laravel переменная `$errors` будет доступна каждый раз, когда вы загружаете представление, даже если оно пустое. Поэтому не нужно проверять с помощью `isset()` ее существование.

Значит, вы можете применить на каждой странице что-то вроде показанного в примере 7.16.

Пример 7.16. Вывод валидационных ошибок

```
@if ($errors->any())
    <ul id="errors">
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
    </ul>
@endif
```

Запросы формы

По мере создания приложения появляются повторяющиеся операции в методах контроллера. Существуют определенные шаблоны, которые повторяются из раза в раз, например проверка входных данных, аутентификация и авторизация

пользователей и возможные перенаправления. Если вам нужна структура, чтобы нормализовать и выделить эти рутинные операции из ваших методов контроллера, помогут запросы форм Laravel.

Запрос формы — это пользовательский класс запроса, предназначенный для сопоставления с отправленной формой. Этот запрос берет на себя ответственность за проверку запроса, авторизацию пользователя и при необходимости перенаправление пользователя при неудаче проверки. Каждый запрос формы (не всегда) явно сопоставляется с одним запросом HTTP — например, «Создать комментарий».

Создание запроса формы

Вы можете создать новый запрос формы из командной строки:

```
php artisan make:request CreateCommentRequest
```

Теперь у вас есть объект запроса формы, доступный по адресу `app/Http/Requests/CreateCommentRequest.php`.

Каждый класс запроса формы предоставляет один-два публичных метода. Первый — `rules()`, который должен возвращать массив правил проверки для этого запроса. Второй (необязательный) метод — `authorize()`. Если он возвращает `true`, то пользователь авторизован для выполнения этого запроса, а если `false` — обращение пользователя отклоняется. В примере 7.17 можно увидеть образец запроса формы.

Пример 7.17. Образец запроса формы

```
<?php

namespace App\Http\Requests;

use App\BlogPost;
use Illuminate\Foundation\Http\FormRequest;

class CreateCommentRequest extends FormRequest
{
    public function authorize()
    {
        $blogPostId = $this->route('blogPost');

        return auth()->check() && BlogPost::where('id', $blogPostId)
            ->where('user_id', auth()->id())->exists();
    }

    public function rules()
    {
        return [
            'body' => 'required|max:1000',
        ];
    }
}
```

Раздел `rules()` в примере 7.17 довольно понятен, но мы кратко рассмотрим `authorize()`.

Извлекаем сегмент из маршрута с именем `blogPost`. Это подразумевает, что определение для этого маршрута выглядит примерно так: `Route::post('blogPosts/blogPost', function() // Действия)`. Как видите, мы назвали параметр маршрута `blogPost`, что делает его доступным в нашем `Request`, при помощи `$this->route('blogPost')`.

Затем мы проверяем, вошел ли пользователь в систему. Если да, существуют ли какие-либо сообщения в блоге с идентификатором, которые принадлежат текущему вошедшему в систему пользователю. Из главы 5 вы узнали о более легких способах проверки прав собственности, теперь немного проясню этот процесс, чтобы сохранить чистоту и простоту. Мы в ближайшее время рассмотрим последствия проверки полномочий. Но важно знать, что возвращение `true` означает, что пользователь авторизован для выполнения указанного действия (в данном случае создания комментария), а `false` — пользователь не авторизован.



Как запросы формы расширяли запрос в пользовательском пространстве до Laravel 5.3

В проектах, работающих на версии Laravel до 5.3, запросы формы расширяли `App\Http\Requests\Request` вместо `Illuminate\Foundation\Http\FormRequest`.

Использование запроса формы

Как мы используем объект запроса формы после его создания? Немного магии Laravel. Любой маршрут (замыкание или метод контроллера), который типизирует запрос формы как один из его параметров, выиграет от определения этого запроса формы.

Попробуем сделать так, как в примере 7.18.

Пример 7.18. Использование запроса формы

```
Route::post('comments', function (App\Http\Requests\CreateCommentRequest $request)
{
    // Сохранение комментария
});
```

Laravel сам вызывает запрос формы. Он проверяет вводимые пользователем данные и авторизует запрос. Если ввод недопустим, он будет действовать так же, как метод `validate()` объекта `Request`, перенаправляя пользователя на предыдущую страницу с сохраненным вводом и соответствующими сообщениями об ошибках. Если пользователь не авторизован, Laravel вернет ошибку 403 Forbidden и не выполнит код маршрута.

Модель массового назначения Eloquent

До сих пор мы рассматривали проверку данных на уровне контроллера, что лучше всего. Вы также можете фильтровать входящие данные на уровне модели.

Это распространенный (но не рекомендуемый) подход — передать весь ввод формы непосредственно в модель базы данных. В Laravel это может выглядеть как в примере 7.19.

Пример 7.19. Передача всей формы в модель Eloquent

```
Route::post('posts', function (Request $request) {
    $newPost = Post::create($request->all());
});
```

Здесь мы предполагаем, что конечный пользователь вежлив, а не злонамерен, и сохранил только те поля, которые мы хотим, чтобы он отредактировал, — возможно, заголовок или тело сообщения.

А если наш конечный пользователь может догадаться или заметить, что у нас есть поле `author_id` в этой таблице `posts`? Если он использует инструменты своего браузера, чтобы добавить поле `author_id` и установить чужой идентификатор, то есть выдать себя за другого человека, создав поддельные сообщения в блогах как принадлежащие этому другому пользователю?

У Eloquent есть концепция, называемая массовым назначением (mass assignment). Она позволяет внести в белый список поля с возможностью заполнения (используя свойство `$fillable` модели) или внести в черный список поля, которые нельзя заполнять (свойством `$guarded` модели), передавая их в массиве для `create()` или `update()`. См. пункт «Массовое назначение» на с. 143 для получения дополнительной информации.

В нашем примере мы могли бы заполнить модель так же, как в примере 7.20, чтобы обеспечить безопасность нашего приложения.

Пример 7.20. Защита модели Eloquent от злонамеренного массового назначения

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    // Отключаем массовое назначение в поле author_id
    protected $guarded = ['author_id'];
}
```

Установив `author_id` в значение `guarded`, мы гарантируем, что злоумышленники больше не смогут переопределять значение этого поля, вручную добавляя его к содержимому формы, которую они отправляют в наше приложение.



Двойная защита с использованием `$request->only()`

Важно хорошо защищать наши модели от массового назначения, но стоит быть осторожным на стороне самого назначения. Вместо использования `$request->all()` рассмотрите возможность применения `$request->only()`, чтобы вы могли указать, какие именно поля вы хотите передать в свою модель:

```
Route::post('posts', function (Request $request) {
    $newPost = Post::create($request->only([
        'title',
        'body',
    ]));
});
```

Синтаксис `{{` и `{!!`

Каждый раз при отображении контента на веб-странице, созданной пользователем, вы должны защищаться от вредоносного ввода, такого как внедрение сценария.

Допустим, вы позволяете пользователям писать сообщения в блоге на вашем сайте. Вероятно, вы не хотите, чтобы они смогли внедрить вредоносный JavaScript, который будет работать в браузерах ничего не подозревающих посетителей. Чтобы этого избежать, нужно экранировать любой пользовательский ввод, который вы показываете на странице.

К счастью, это почти полностью сделано за вас. При использовании шаблонизатора Blade от Laravel по умолчанию синтаксис «вывода» (`{{ $stuffToEcho }}`) пропускает вывод через `htmlentities()` автоматически. Это лучший способ РНР сделать пользовательский контент безопасным для вывода. Нужно постараться, чтобы избежать экранирования, применяя синтаксис `{!! $stuffToEcho !!}`.

Тестирование

Для тестирования ваших взаимодействий с пользовательским вводом нужно смоделировать допустимый и недопустимый вводы пользователя и обеспечить условия: если ввод недопустимый, пользователь перенаправляется, а если ввод действителен, он попадает в правильное место (например, в базу данных).

Комплексное тестирование приложений Laravel упрощает эту задачу.



Для Laravel версии выше 5.4 требуется BrowserKit

Если вы хотите работать с определенными взаимодействиями с пользователями на странице и в ваших формах в Laravel 5.4 или более поздней версии, стоит использовать пакет тестирования Laravel BrowserKit. Загрузите этот пакет:

```
composer require laravel/browser-kit-testing --dev
```

Измените базовый класс TestCase вашего приложения так, чтобы он расширял `Laravel\BrowserKitTesting\TestCase` вместо `Illuminate\Foundation\Testing\TestCase`.

Начнем с неверного маршрута, который, как мы ожидаем, будет отклонен, как в примере 7.21.

Пример 7.21. Проверка того, что недопустимый ввод отклонен

```
public function test_input_missing_a_title_is_rejected()
{
    $response = $this->post('posts', ['body' => 'This is the body of my post']);
    $response->assertRedirect();
    $response->assertSessionHasErrors();
}
```

Здесь мы задаем, чтобы после неверного ввода пользователь перенаправлялся в нужное место (с прикрепленными ошибками). Мы используем несколько пользовательских функций PHPUnit, которые Laravel добавляет сюда.



Различные названия для методов тестирования до Laravel 5.4

До Laravel 5.4 функция `assertRedirect()` называлась `assertRedirectedTo()`.

Как же мы можем проверить правильность нашего маршрута? Посмотрите пример 7.22.

Пример 7.22. Проверка того, что допустимый ввод будет обрабатываться

```
public function test_valid_input_should_create_a_post_in_the_database()
{
    $this->post('posts', ['title' => 'Post Title', 'body' => 'This is the body']);
    $this->assertDatabaseHas('posts', ['title' => 'Post Title']);
}
```

Если вы тестируете что-либо с использованием базы данных, нужно больше узнать о миграциях и транзакциях БД. Подробнее об этом — в главе 12.



Различные названия для методов тестирования до Laravel 5.4

В проектах, которые работают с версиями Laravel до 5.4, `assertDatabaseHas()` следует заменить на `seeInDatabase()`.

Резюме

Есть много способов получить одни и те же данные: с помощью фасада `Request`, глобального хелпера `request()` или внедрения экземпляра `Illuminate\Http\Request`. Каждый из них помогает получить все/некоторые входные данные или определенные фрагменты данных. Возможно, могут быть некоторые особенности при работе с файлами и входными данными JSON.

Сегменты пути URI — возможные источники пользовательского ввода. Они также доступны с помощью инструментов запросов.

Валидацию можно выполнить вручную с помощью `Validator::make()` и автоматически методом `validate()` или запросами формы. Каждый автоматический инструмент после неудачной валидации перенаправляет пользователя на предыдущую страницу со всеми сохраненными старыми данными и сообщениями об ошибках.

Представления и модели Eloquent также должны быть защищены от вредоносного ввода пользователя. Защитите представления Blade синтаксисом двойной фигурной скобки (`{{ }}`), что экранирует ввод данных пользователем. Обезопасьте модели, передавая только определенные поля в групповые методы, используя `$request->only()` и определяя правила массового назначения в самой модели.

8

Интерфейсы Artisan и Tinker

С начала установки современные PHP-фреймворки готовы к множеству взаимодействий в командной строке. Для этого Laravel предоставляет три основных инструмента: Artisan — набор встроенных функций командной строки с возможностью добавления новых; Tinker, REPL или интерактивную оболочку для вашего приложения; установщик, который мы уже рассмотрели в главе 2.

Введение в интерфейс Artisan

Вы уже научились использовать команды Artisan. Они выглядят примерно так:

```
php artisan make:controller PostsController
```

В корневой папке своего приложения вы увидите, что `artisan` на самом деле является файлом PHP. Поэтому вы начинаете вызов с `php artisan` и передаете этот файл в PHP для разбора. Все, что находится после этого, просто отправляется в Artisan в качестве аргументов.



Синтаксис консоли Symfony

Artisan фактически является уровнем поверх компонента консоли Symfony (<http://bit.ly/2fVqOT8>). Поэтому, если вы знакомы с введением команд консоли Symfony, почувствуйте себя как дома.

Список команд Artisan для приложения может меняться в зависимости от пакета или конкретного кода приложения. Стоит проверять каждое новое приложение, с которым вы сталкиваетесь, чтобы увидеть доступные команды.

Для этого можно запустить `php artisan list` из корня проекта. Если вы просто запустите `php artisan` без параметров, будет то же самое.

Основные команды Artisan

Здесь недостаточно места, чтобы описать все команды Artisan, но мы рассмотрим многие из них. Начнем с основных.

- ❑ **clear-compiled.** Удаляет файл скомпилированного класса Laravel, который похож на внутренний кэш Laravel. Применяйте, когда что-то не так, но вы не знаете почему.
- ❑ **down, up.** Переводит ваше приложение в режим обслуживания, чтобы вы могли исправить ошибку, запустить миграции или сделать что-то еще и восстановить приложение из режима обслуживания соответственно.
- ❑ **dump-server (5.7+).** Запускает сервер дампа (см. раздел «Сервер дампа Laravel» на с. 231) для сбора и вывода загруженных переменных.
- ❑ **env.** Показывает, в какой среде Laravel работает в данный момент. Эквивалентно выводу `app()->environment()` в приложении.
- ❑ **help.** Предоставляет помощь для команды, например, `php artisan help commandName`.
- ❑ **migrate.** Запускает все миграции базы данных.
- ❑ **optimize.** Очищает и обновляет файлы конфигурации и маршрута.
- ❑ **preset.** Меняет кодогенерацию клиентской части на другую.
- ❑ **serve.** Закрепляет PHP-сервер за адресом `localhost:8000`. Можно настроить хост и/или порт с помощью `--host` и `--port`.
- ❑ **tinker.** Запускает Tinker REPL, о котором я расскажу позже в этой главе.



Изменения в списке команд Artisan с течением времени

Список команд Artisan и их названия незначительно изменились за время существования Laravel. Я постараюсь отметить, когда они меняются, но пока здесь все актуально для Laravel 5.8. Если вы не работаете в 5.8, лучший способ узнать, что вам доступно, — запустить `php artisan` из вашего приложения.

Параметры

Рассмотрим несколько важных параметров, которые можно использовать в любом вызове команды Artisan.

- ❑ **-q** — подавить весь вывод.
- ❑ **-v, -vv и -vvv** — установить уровень детализации вывода (нормальный, подробный и отладочный).
- ❑ **--no-interaction** — подавляет интерактивные вопросы, поэтому команда не будет прерывать автоматизированные процессы, выполняющие ее.
- ❑ **--env** — позволяет задать, в какой среде должна работать команда Artisan (локальная, производственная и т. д.).
- ❑ **--version** — показывает, на какой версии Laravel работает ваше приложение.

Команды Artisan во многом подобны базовым командам оболочки: вы можете запускать их вручную, но они также могут функционировать как часть некоторого автоматизированного процесса.

Например, есть множество процессов автоматического развертывания, которым выгодны определенные команды Artisan. Возможно, вы захотите запускать `php artisan config:cache` каждый раз, когда развертываете приложение. Флаги, такие как `-q` и `--no-interaction`, обеспечивают бесперебойную работу сценариев развертывания без участия человека.

Сгруппированные команды

Остальные команды, доступные сразу же «из коробки», сгруппированы по контексту. Мы не рассмотрим здесь все, но я расскажу про контекст каждой.

- ❑ **app** — здесь просто содержится `app:name`, позволяющий заменить каждый экземпляр стандартного верхнего уровня пространства имен `App\` пространством имен по вашему выбору. Например, приложение `php artisan app:name MyApplication`. Я рекомендую избегать этой функции и сохранять корневое пространство имен вашего приложения как `App`.
- ❑ **auth** — здесь есть только `auth:clear-resets`, который обновляет все токены сброса пароля с истекшим сроком действия в базе данных.
- ❑ **cache** — `cache:clear` очищает кэш, `cache:forget` удаляет отдельный элемент из кэша, а `cache:table` создает миграцию базы данных, если вы планируете использовать драйвер кэша `database`.
- ❑ **config** — `config:cache` кэширует ваши настройки конфигурации для более быстрого поиска. Чтобы очистить кэш, используйте `config:clear`.
- ❑ **db** — `db:seed` заполняет вашу базу данных, если вы настроили наполнители БД.
- ❑ **event** — `event:generate` создает файлы пропущенных событий и слушателей событий на основе определений в `EventServiceProvider`. Вы узнаете больше о событиях в главе 16.
- ❑ **key** — `key:generate` создаст случайный ключ шифрования приложения в вашем файле `.env`.



Повторный запуск `artisan key:generate` означает потерю некоторых зашифрованных данных

Если вы запустите `php artisan key:generate` в своем приложении несколько раз, то каждый вошедший в данный момент пользователь будет выброшен из системы. Кроме того, любые зашифрованные вручную данные больше не будут расшифровываться. Чтобы узнать больше, ознакомьтесь со статьей «APP_KEY и вы» моего коллеги из TightenIt Джейка Бэтмена по адресу <http://bit.ly/2U972qd>.

- ❑ **make** — **make:auth** переделывает представления и соответствующие маршруты для целевой страницы, панели пользователя и страниц входа и регистрации.

Все остальные действия **make:** создают отдельный элемент и имеют параметры, которые соответственно меняются. Чтобы узнать больше о параметрах любой отдельной команды, используйте **help** соответствующей команды.

Например, можно запустить **php artisan help make:migration** и узнать, что вы можете передать **--create=tableNameHere**, чтобы создать миграцию, в которой уже есть синтаксис создания таблицы, как показано здесь: **php artisan make:migration create_posts_table --create=posts**.

- ❑ **migrate** — нужна для запуска всех миграций, но есть несколько других связанных с миграцией команд. Вы можете создать таблицу **migrations** (чтобы отслеживать выполненные миграции) с помощью **migrate:install**, отменить ваши миграции и начать с нуля — **migrate:reset**, сбросить ваши миграции и снова запустить их, используя **migrate:refresh**, откатить только одну миграцию с помощью **migrate:rollback**, удалить все таблицы и повторно запустить все миграции, применив **migrate:fresh**, или проверить состояние ваших миграций, написав **migrate:status**.
- ❑ **notifications** — **notifications:table** генерирует миграцию, которая создает таблицу для уведомлений базы данных.
- ❑ **package**.

5.3 В версиях Laravel до 5.5 включение нового специфичного для Laravel пакета в ваше приложение требует его регистрации вручную в **config/app.php**. Однако в 5.5 для Laravel возможно автоматическое обнаружение этих пакетов. **package:discover** перестраивает «обнаруженный» манифест сервис-провайдеров Laravel из ваших внешних пакетов.

- ❑ **queue** — я расскажу об очередях Laravel в главе 16, но основная идея в том, что вы можете помещать задания работнику в удаленные очереди для выполнения по порядку. Эта группа команд предоставляет все инструменты взаимодействия с вашими очередями, такие как **queue:listen** для начала прослушивания очереди, **queue:table** для создания миграции для очередей, поддерживаемых базой данных, и **queue:flush** для сброса всех неудачных заданий очереди. Есть еще несколько в главе 16.
- ❑ **route** — если вы запустите **route:list**, то увидите определения каждого маршрута, заданного в приложении, включая команду/команды каждого маршрута, путь, имя, действие контроллера/замыкания и middleware. Вы можете кэшировать определения маршрутов для более быстрого поиска с помощью **route:cache** и очищать кэш с помощью **route:clear**.
- ❑ **schedule** — я расскажу о cron-подобном планировщике Laravel в главе 16, но для его работы необходимо настроить системный cron на запуск **schedule:run** раз в минуту:

```
* * * * * php /home/myapp.com/artisan schedule:run >> /dev/null 2>&1
```

Эта команда Artisan предназначена для регулярного запуска, чтобы включить основной сервис Laravel.

- ❑ **session** — `session:table` создает миграцию для приложений, использующих сессии с поддержкой базы данных.
- ❑ **storage** — `storage:link` создает символическую ссылку из `public/storage` в `storage/app/public`. Это общее соглашение в приложениях Laravel, позволяющее легко размещать пользовательские загрузки (или другие файлы, которые обычно попадают в `storage/app`) там, где они будут достижимы по публичному URL-адресу.
- ❑ **vendor** — некоторые специфичные для Laravel пакеты должны «публиковать» отдельные из своих ресурсов, чтобы они могли обслуживаться из вашего каталога `public` или для изменения. В любом случае эти пакеты регистрируют эти «публичные ресурсы» в Larave. Когда вы запускаете `vendor:publish`, он публикует их в специфичных для них местах.
- ❑ **view** — движок визуализации Laravel автоматически кэширует ваши представления. Обычно он хорошо справляется со своей неспособностью кэширования, но если вы когда-нибудь заметите, что он застрял, запустите `view:clear`, чтобы очистить кэш.

Написание пользовательских команд Artisan

Теперь, когда мы рассмотрели команды Artisan, поставляемые с Laravel изначально, поговорим о написании собственных команд.

Для этого есть команда Artisan. Команда `php artisan make:command YourCommandName` создаст новую команду Artisan в `app/Console/Commands/{YourCommandName}.php`.



php artisan make:command

53 Сигнатура команды `make:command` менялась несколько раз. Первоначально это была команда `command:make`, но некоторое время в версии 5.2 она называлась `console:make`, а затем `make:console`.

Наконец, в 5.3 все было решено: все генераторы находятся в пространстве имен `make:`, и теперь для создания новых команд Artisan используется команда `make:command`.

Ваш первый аргумент должен быть именем класса команды, и вы можете при желании передать параметр `--command`, чтобы определить, какой будет команда терминала (например, `appname:action`). Попробуем:

```
php artisan make:command WelcomeNewUsers --command=email:newusers
```

В примере 8.1 показан результат.

Пример 8.1. Образец команды Artisan по умолчанию

```
<?php
```

```
namespace App\Console\Commands;
```

```
use Illuminate\Console\Command;
```

```
class WelcomeNewUsers extends Command
```

```
{
    /**
     * Имя и сигнатура консольной команды
     *
     * @var string
     */
    protected $signature = 'email:newusers';

    /**
     * Описание консольной команды
     *
     * @var string
     */
    protected $description = 'Command description';

    /**
     * Создание нового экземпляра команды
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Выполнить консольную команду
     *
     * @return mixed
     */
    public function handle()
    {
        //
    }
}
```

Легко определить сигнатуру команды, текст помощи, который она показывает в списках команд, и поведение команды при создании экземпляра (`__construct()`) и выполнении (`handle()`).



Связывание команд вручную до Laravel 5.5

В проектах с версиями Laravel до 5.5 команды должны были быть связаны вручную в `app\Console\Kernel.php`. Если ваше приложение использует более старую версию Laravel, добавьте полное имя класса для вашей команды в массив `$commands` в этом файле, и он будет зарегистрирован:

```
protected $commands = [
    \App\Console\Commands\WelcomeNewUsers::class,
];
```

Пример команды

Мы еще не рассмотрели почту или Eloquent в этой главе (почта описана в главе 15, а Eloquent — в главе 5), но образец метода `handle()` в примере 8.2 должен читаться довольно четко.

Пример 8.2. Образец метода `handle()` для обработки команды Artisan

```
...
class WelcomeNewUsers extends Command
{
    public function handle()
    {
        User::signedUpThisWeek()->each(function ($user) {
            Mail::to($user)->send(new WelcomeEmail);
        });
    }
}
```

Теперь каждый раз при запуске `php artisan email:newusers` эта команда будет находить всех подписавшихся на этой неделе пользователей и отправлять им приветственные письма.

Если вы предпочитаете внедрять свои почтовые и пользовательские зависимости вместо использования фасадов, можно определить их тип в конструкторе команд. И контейнер Laravel внедрит их для вас при создании экземпляра команды.

Взгляните на пример 8.3 — так может выглядеть пример 8.2, но с внедрением зависимостей и извлечением его поведения в класс обслуживания.

Пример 8.3. Та же команда, рефакторинг

```
...
class WelcomeNewUsers extends Command
{
    public function __construct(UserMailer $userMailer)
    {
        parent::__construct();
```

```
$this->userMailer = $userMailer  
}  
  
public function handle()  
{  
    $this->userMailer->welcomeNewUsers();  
}
```

НЕ УСЛОЖНЯЙТЕ

Можно вызывать команды Artisan из остальной части вашего кода для инкапсуляции частей логики приложения.

Однако вместо этого в документации по Laravel рекомендуется упаковать логику приложения в класс сервиса и внедрить его в вашу команду. Консольные команды похожи на контроллеры: не являются классами домена; они регуляризовщики, которые лишь правильно направляют поступающие запросы.

Аргументы и параметры

Свойство `$signature` новой команды выглядит так, будто оно может содержать только имя команды. Но также это свойство будет участвовать там, где вы будете определять любые аргументы и опции для команды. Существует специальный простой синтаксис, который можно использовать для добавления аргументов и параметров в ваши команды Artisan.

Прежде чем мы углубимся в этот синтаксис, взглянем на пример для некоторого контекста:

```
protected $signature = 'password:reset {userId} [--sendEmail]';
```

Аргументы: обязательные, необязательные и/или со значениями по умолчанию

Чтобы определить обязательный аргумент, поместите его в фигурные скобки:

```
password:reset {userId}
```

Для необязательного аргумента добавьте знак вопроса:

```
password:reset {userId?}
```

Чтобы сделать аргумент необязательным и задать значение по умолчанию, используйте следующую конструкцию:

```
password:reset {userId=1}
```

Опции: обязательные значения, значения по умолчанию и сокращенные формы

Опции похожи на аргументы, но имеют префикс `--` и могут использоваться без значения. Чтобы добавить базовую опцию, заключите ее в фигурные скобки:

```
password:reset {userId} {--sendEmail}
```

Если опция требует значение — добавьте `=` к ее сигнатуре:

```
password:reset {userId} {--password=}
```

Если вы хотите передать значение по умолчанию, добавьте его после `=`:

```
password:reset {userId} {--queue=default}
```

Аргументы и опции массива

Для аргументов опций используйте символ `*`, если вы хотите принять массив в качестве входных данных:

```
password:reset {userIds*}  
password:reset {--ids=}
```

Применение аргументов и опций массива выглядит как пример 8.4.

Пример 8.4. Использование синтаксиса массива с командами Artisan

```
// Аргумент  
php artisan password:reset 1 2 3  
  
// Параметр  
php artisan password:reset --ids=1 --ids=2 --ids=3
```



Аргументы массива должны быть последним аргументом

Поскольку аргумент массива захватывает каждый параметр после его определения и добавляет эти параметры как элементы массива, аргумент массива должен быть последним аргументом в сигнатуре команды Artisan.

Описания ввода

Встроенные команды Artisan могут дать нам больше информации об их параметрах, если мы используем `artisan help`. Мы можем предоставить ту же информацию о наших пользовательских командах. Просто добавьте двоеточие и текст описания в фигурных скобках, как в примере 8.5.

Пример 8.5. Определение текста описания для аргументов и опций Artisan

```
protected $signature = 'password:reset
                        {userId : The ID of the user}
                        {--sendEmail : Whether to send user an email}';
```

Использование ввода

Теперь, когда мы запросили этот ввод, как мы используем его в методе `handle()` нашей команды? У нас есть два набора методов для получения значений аргументов и параметров.

Методы `argument()` и `arguments()`

`$this->arguments()` возвращает массив всех аргументов (первым элементом здесь будет имя команды). `$this->argument()`, вызываемый без параметров, возвращает тот же ответ; метод множественного числа, который я предпочитаю, лучше читается и доступен только после Laravel 5.3.

Чтобы получить значение только одного аргумента, передайте имя аргумента в качестве параметра в `$this->argument()`, как показано в примере 8.6.

Пример 8.6. Использование `$this->arguments()` в команде Artisan

```
// С определением "password:reset {userId}"
php artisan password:reset 5

// $this->arguments() возвращает этот массив
[
    "command": "password:reset",
    "userId": "5",
]

// $this->argument('userId') возвращает эту строку
"5"
```

Методы `option()` и `options()`

`$this->options()` возвращает массив всех опций, включая те, которые по умолчанию будут `false` или `null`. `$this->option()`, вызываемый без параметров, возвращает тот же ответ. Предпочитаемый мной метод множественного числа лучше читается и доступен только после Laravel 5.3.

Чтобы получить значение только одной опции, передайте имя аргумента в качестве параметра в `$this->option()`, как показано в примере 8.7.

Пример 8.7. Использование `$this->options()` в команде Artisan

```
// С определением "password:reset {--userId=}"
php artisan password:reset --userId=5

// $this->options() возвращает этот массив
[
    "userId" => "5",
    "help" => false,
    "quiet" => false,
    "verbose" => false,
    "version" => false,
    "ansi" => false,
    "no-ansi" => false,
    "no-interaction" => false,
    "env" => null,
]

// $this->option('userId') возвращает эту строку
"5"
```

В примере 8.8 показана команда Artisan, использующая `argument()` и `option()` в методе `handle()`.

Пример 8.8. Получение ввода от команды Artisan

```
public function handle()
{
    // Все аргументы, включая имя команды
    $arguments = $this->arguments();

    // Только аргумент 'userId'
    $userid = $this->argument('userId');

    // Все опции, включая некоторые по умолчанию вроде 'no-interaction' и 'env'
    $options = $this->options();

    // Только опция 'sendEmail'
    $sendEmail = $this->option('sendEmail');
}
```

Приглашения

Есть еще несколько способов получить пользовательский ввод из вашего кода `handle()`, и все они включают в себя запрос пользователю ввести информацию во время выполнения вашей команды.

- ❑ `ask()`. Предлагает пользователю ввести произвольный текст:

```
$email = $this->ask('What is your email address?');
```

- ❑ `secret()`. Предлагает пользователю ввести произвольный текст, но скрывает ввод с помощью звездочек:

```
$password = $this->secret('What is the DB password?');
```

- ❑ `confirm()`. Запрашивает у пользователя ответ «да/нет» и возвращает логическое значение:

```
if ($this->confirm('Do you want to truncate the tables?')) {
    //
}
```

Все ответы, кроме у или У, будут рассматриваться как «нет».

- ❑ `anticipate()`. Предлагает пользователю ввести произвольный текст и предлагает варианты автозаполнения. Тем не менее позволяет пользователю печатать все, что он захочет:

```
$album = $this->anticipate('What is the best album ever?', [
    'The Joshua Tree', 'Pet Sounds', 'What's Going On'
]);
```

- ❑ `choice()`. Предлагает пользователю выбрать один из предоставленных вариантов. Последний параметр является значением по умолчанию, если пользователь не сделал выбор:

```
$winner = $this->choice(
    'Who is the best football team?',
    ['Gators', 'Wolverines'],
    0
);
```

Последний параметр по умолчанию должен быть ключом массива. Поскольку мы передали неассоциативный массив, ключ для `Gators` равен `0`. Вы также можете по желанию указать свой массив:

```
$winner = $this->choice(
    'Who is the best football team?',
    ['gators' => 'Gators', 'wolverines' => 'Wolverines'],
    'gators'
);
```

Вывод

Во время выполнения вашей команды можно написать сообщения пользователю. Самый легкий способ — использовать `$this->info()` для вывода простого текста зеленого цвета:

```
$this->info('Your command has run successfully.');
```

Вам также доступны методы `comment()` (оранжевый), `question()` (жирный сине-зеленый), `error()` (жирный красный) и `line()` (бесцветный).

Обратите внимание, что точные цвета могут варьироваться от машины к машине, но все пытаются соответствовать стандартам для связи с конечным пользователем.

Табличный вывод

Метод `table()` упрощает создание таблиц ASCII, заполненных вашими данными. Посмотрите на пример 8.9.

Пример 8.9. Вывод таблиц с помощью команд Artisan

```
$headers = ['Name', 'Email'];

$data = [
    ['Dhriti', 'dhriti@amrit.com'],
    ['Moses', 'moses@gutierrez.com'],
];

// Или вы можете получить аналогичные данные из базы данных:
$data = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $data);
```

В примере 8.9 есть два набора данных: заголовки и сами данные. Оба содержат две «ячейки» на «строку». Первая ячейка в каждой строке — это имя, а вторая — адрес электронной почты. Таким образом, данные из вызова Eloquent (который ограничивается извлечением только имени и адреса электронной почты) совпадают с заголовками.

В примере 8.10 показан вывод таблицы.

Пример 8.10. Пример вывода таблицы Artisan

```
+-----+-----+
| Name   | Email                               |
+-----+-----+
| Dhriti | dhriti@amrit.com                   |
| Moses  | moses@gutierrez.com               |
+-----+-----+
```

Индикаторы выполнения

Если вы когда-либо запускали `npm install`, то видели индикатор выполнения. Создадим один в примере 8.11.

Пример 8.11. Примерный индикатор выполнения Artisan

```
$totalUnits = 10;
$this->output->progressStart($totalUnits);

for ($i = 0; $i < $totalUnits; $i++) {
    sleep(1);

    $this->output->progressAdvance();
}

$this->output->progressFinish();
```

Сначала мы сообщили системе, сколько «единиц» нужно проработать. Может быть, «единица» — это пользователь, а у вас 350 пользователей. Тогда индикатор раз- делит доступную ширину окна на 350 и будет увеличивать ее на 1/350 при каждом запуске `progressAdvance()`. Когда вы закончите, запустите `progressFinish()`, чтобы индикатор знал, что он отображен.

Команды на основе замыканий

5.3 Для упрощения процесса определения команд можно написать команды как замыкания, а не как классы, определяя их в `routes/console.php`. Все, что мы обсудим в этой главе, будет применяться одинаково, но вы определите и зарегистрируете команды за один шаг в этом файле, как показано в примере 8.12.

Пример 8.12. Определение команды Artisan с использованием замыкания

```
// routes/console.php
Artisan::command(
    'password:reset {userId} [--sendEmail]',
    function ($userId, $sendEmail) {
        $userId = $this->argument('userId');
        // Некий код...
    }
);
```

Вызов команд Artisan в нормальном коде

Хотя команды Artisan предназначены для запуска из командной строки, можно вызывать их из другого кода.

Самый простой способ — использовать фасад **Artisan**. Вы можете вызвать команду с помощью `Artisan::call()` (которая будет возвращать код завершения команды) или поставить команду в очередь, использовав `Artisan::queue()`.

Обе команды принимают два параметра: команда терминала (`password:reset`) и массив параметров для его передачи. В примере 8.13 показана работа с аргументами и опциями.

Пример 8.13. Вызов команд Artisan из другого кода

```
Route::get('test-artisan', function () {
    $exitCode = Artisan::call('password:reset', [
        'userId' => 15,
        '--sendEmail' => true,
    ]);
});
```

Аргументы передаются путем ввода имени аргумента, а опции без значения могут передаваться как `true` или `false`.



В Laravel 5.8+ вы можете вызывать команды Artisan из своего кода гораздо проще. Передайте ту же строку, которую вы вызываете из командной строки, в `Artisan::call()`:

```
Artisan::call('password:reset 15 --sendEmail')
```

Вы также можете вызывать команды Artisan из других, используя `$this->call()` (то же самое, что и `Artisan::call()`), или `$this->callSilent()`, что аналогично, но подавляет весь вывод. Посмотрите на пример 8.14.

Пример 8.14. Вызов команд Artisan из других команд Artisan

```
public function handle()
{
    $this->callSilent('password:reset', [
        'userId' => 15,
    ]);
}
```

Наконец, вы можете внедрить экземпляр контракта `Illuminate\Contracts\Console\Kernel` и использовать его метод `call()`.

Tinker

Tinker — это цикл REPL, или «чтение — выполнение — распечатка». REPL работает так же, как IRB в Ruby.

REPL выводит среду выполнения, похожую на среду выполнения командной строки, которое имитирует состояние ожидания вашего приложения. Вы вводите свои команды в REPL, нажимаете **Return** и ожидаете, что в ответ будет получен результат и распечатан ответ.

В примере 8.15 приведен короткий образец, чтобы дать представление о том, как он работает и насколько может быть полезен. Мы входим в REPL, введя `php artisan tinker`, получаем пустое приглашение (`>>>`). Каждый ответ на наши команды печатается в строке с предваряющим знаком `=>`.

Пример 8.15. Использование Tinker

```
$ php artisan tinker
```

```
>>> $user = new App\User;
=> App\User: {}
>>> $user->email = 'matt@mattstauffer.com';
```

```
=> "matt@mattstauffer.com"
>>> $user->password = bcrypt('superSecret');
=> "$2y$10$TWPgBC7e8d1bvJ1q5kv.VDUGfYDnE9gAN14m1euB3htIY2dxcQfQ5"
>>> $user->save();
=> true
```

Мы создали нового пользователя, установили некоторые данные (для безопасности хешировали пароль с помощью `bcrypt()`) и сохранили их в базе данных. И это реально. Будь это производственное приложение, мы бы просто создали нового пользователя в системе.

Это делает Tinker отличным инструментом для простого взаимодействия с базой данных, тестирования новых идей и запуска фрагментов кода, когда трудно найти место для размещения их в исходных файлах приложения.

Tinker работает на Psy Shell (<http://psysh.org/>). Посмотрите остальные возможности Tinker.

Сервер дампа Laravel

Одним из распространенных методов отладки состояния ваших данных во время разработки является использование хелпера `dump()` Laravel, который запускает более красивый `var_dump()` для всего, что вы ему передаете. Это хорошо, но часто могут возникать проблемы с представлениями.

57 В проектах под управлением Laravel 5.7 и более поздних версий можно включить сервер дампа Laravel, который ловит эти операторы `dump()` и отображает их вывод в консоли вместо отображения на страницу.

Чтобы запустить сервер дампа в локальной консоли, перейдите в корневой каталог проекта и запустите `php artisan dump-server`:

```
$ php artisan dump-server
```

```
Laravel Var Dump Server
=====
```

```
[OK] Server listening on tcp://127.0.0.1:9912
```

```
// Выйти с сервера можно, нажав CTRL+C.
```

Теперь попробуйте где-нибудь использовать хелпер `dump()` в вашем коде. Для проверки введите этот код в ваш файл `route/web.php`:

```
Route::get('/', function () {
    dump('Dumped Value');

    return 'Hello World';
});
```

Без сервера дампов вы увидите и дампы, и свой **Hello, World**. Но с запущенным сервером дампов вы увидите только **Hello, World** в браузере. В вашей консоли показано, что сервер дампа перехватил этот `dump()`, и вы можете проверить это:

```
GET http://myapp.test/
```

```
-----
```

```
-----
date           Tue, 18 Sep 2018 22:43:10 +0000
controller     "Closure"
source         web.php on line 20
file           routes/web.php
-----
```

```
"Dumped Value"
```

Тестирование

Поскольку вы знаете, как вызывать команды Artisan из кода, это легко сделать в тесте и убедиться, что любое ожидаемое поведение было правильным, как в примере 8.16. В наших тестах мы используем `$this->artisan()` вместо `Artisan::call()`, потому что он имеет тот же синтаксис, но добавляет несколько связанных с тестированием операторов.

Пример 8.16. Вызов команд Artisan из теста

```
public function test_empty_log_command_emptyies_logs_table()
{
    DB::table('logs')->insert(['message' => 'Did something']);
    $this->assertCount(1, DB::table('logs')->get());

    $this->artisan('logs:empty'); // Same as Artisan::call('logs:empty');
    $this->assertCount(0, DB::table('logs')->get());
}
```

5.7 В проектах под управлением Laravel 5.7 и более поздних версий можно связать несколько новых операторов с вашими вызовами `$this->artisan()`. Они упрощают тестирование команд Artisan — не только влияние, которое они оказывают на остальную часть вашего приложения, но и то, как они на самом деле работают. В примере 8.17 приведен образец такого синтаксиса.

Пример 8.17. Обработка ввода и вывода команд Artisan

```
public function testItCreatesANewUser()
{
    $this->artisan('myapp:create-user')
        ->expectsQuestion("What's the name of the new user?", "Wilbur Powery")
        ->expectsQuestion("What's the email of the new user?", "wilbur@thisbook.co")
        ->expectsQuestion("What's the password of the new user?", "secret")
        ->expectsOutput("User Wilbur Powery created!");
}
```



```
$this->assertDatabaseHas('users', [  
    'email' => 'wilbur@thisbook.co'  
]);  
}
```

Резюме

Команды Artisan являются инструментами командной строки Laravel. Laravel поставляется с несколькими готовыми заранее командами Artisan, но их легко создавать и вызывать из командной строки или собственного кода.

Tinker — это REPL, который упрощает вход в среду вашего приложения и взаимодействие с реальными кодом и данными, а сервер дампа позволяет отлаживать код без остановки его выполнения.

9

Аутентификация и авторизация пользователей

Настройка базовой системы аутентификации пользователей — включая регистрацию и вход в систему, сессии, сброс паролей и права доступа — часто является одним из наиболее трудоемких этапов разработки «фундамента» приложения. Это главный кандидат на извлечение функциональности в библиотеку, а таких библиотек довольно много.

Однако в силу того, что потребности разных проектов в плане аутентификации могут сильно различаться, системы аутентификации обычно быстро становятся громоздкими и неудобными в использовании. К счастью, Laravel позволяет создать систему аутентификации, которая будет не только простой в использовании и понимании, но и достаточно гибкой для удовлетворения различных потребностей.

Каждая новая установка Laravel уже изначально содержит миграцию `create_users_table` и модель `User`. Laravel предлагает команду `Artisan make:auth`, которая выполняет начальное заполнение коллекции представлений и маршрутов, связанных с аутентификацией. В состав каждой установки также входят контроллеры `RegisterController`, `LoginController`, `ForgotPasswordController` и `ResetPasswordController`. Эти простые и понятные API вместе с принятыми соглашениями позволяют создать эффективную систему аутентификации и авторизации.



Различия в структуре контроллеров аутентификации в Laravel до версии 5.3



В Laravel 5.1 и 5.2 большая часть данной функциональности находилась в контроллере `AuthController`. В версии 5.3 и выше данная функциональность была разбита на несколько контроллеров. Многие из того, что здесь будет сказано относительно настройки маршрутов перенаправления, гардов аутентификации и тому подобного, несколько различается в версиях 5.1 и 5.2 (хотя все основные функции одинаковы). Таким образом, если вы используете версию 5.1 или 5.2 и вас не совсем устраивает то, как ведет себя аутентификация по умолчанию, следует чуть подробнее изучить контроллер `AuthController` и посмотреть, какие именно изменения необходимо внести.

Модель User и миграция

Первое, что вы увидите при создании нового приложения Laravel, — это миграция `create_users_table` и модель `App\User`. Приведенный в примере 9.1 код миграции показывает, какие поля будут созданы в таблице `users`.

Пример 9.1. Миграция для таблицы пользователей, предоставляемая фреймворком Laravel по умолчанию

```
Schema::create('users', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('name');
    $table->string('email')->unique();
    $table->string('password');
    $table->rememberToken();
    $table->timestamps();
});
```

Здесь автоинкрементный первичный ключ — идентификатор, имя, уникальный адрес электронной почты, пароль, токен «Запомнить меня», а также временные метки создания и изменения. Этого вполне достаточно для реализации базовых функций аутентификации пользователей в большинстве приложений.



Различие между аутентификацией и авторизацией

Аутентификация — подтверждение личности пользователя с предоставлением ему права на соответствующие действия в системе. Это включает в себя вход/выход в системе, а также любые процедуры идентификации пользователей во время использования ими приложения.

Авторизация — это проверка на разрешение аутентифицированному пользователю выполнять определенные действия (то есть авторизован ли он для их выполнения). Например, с помощью системы авторизации можно запретить доступ к информации о прибыльности сайта для всех, кроме администраторов.

Модель `User` выглядит немного сложнее, как вы можете видеть в примере 9.2. Хотя класс `App\User` является простым, он расширяет класс `Illuminate\Foundation\Auth\User`, который подгружает несколько трейтов (trait).

Пример 9.2. Модель `User`, предоставляемая фреймворком Laravel по умолчанию

```
<?php
// App\User

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
```

```
class User extends Authenticatable
{
    use Notifiable;

    /**
     * Массив присваиваемых атрибутов.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * Атрибуты, которые должны быть скрыты для массивов.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    /**
     * Атрибуты, которые должны быть приведены к нативным типам.
     *
     * @var array
     */
    protected $casts = [
        'email_verified_at' => 'datetime',
    ];
}

<?php
// Illuminate\Foundation\Auth\User

namespace Illuminate\Foundation\Auth;

use Illuminate\Auth\Authenticatable;
use Illuminate\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Foundation\Auth\Access\Authorizable;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\Access\Authorizable as AuthorizableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements
    AuthenticatableContract,
    AuthorizableContract,
    CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword, MustVerifyEmail;
}
```



Освежите в памяти сведения о моделях Eloquent

Если этот код кажется вам незнакомым, стоит освежить в памяти материал главы 5 и только потом продолжить изучение того, как работают модели Eloquent.

Что можно понять из этой модели? Пользователи размещаются в таблице `users`; это имя применяется в Laravel в соответствии с названием класса. При создании нового пользователя можно заполнить свойства `name`, `email`, `password`, и свойства `password`, `remember_token` исключаются при выводе данных о пользователях в формате JSON. Пока все выглядит замечательно.

Исходя из того, какими контрактами и трейтами обладает расширенный класс `Illuminate\Foundation\Auth\User`, можно заключить, что некоторые возможности фреймворка (аутентификация, авторизация и сброс паролей) могут применяться к другим моделям, помимо `User`, как по отдельности, так и вместе.

КОНТРАКТЫ И ИНТЕРФЕЙСЫ

Я употребляю слово «контракт», а иногда — слово «интерфейс», и почти все интерфейсы в Laravel находятся в пространстве имен `Contracts`.

Интерфейс языка PHP, по сути, соглашение между двумя классами о том, что один из них будет вести себя определенным образом. Это соглашение напоминает контракт между классами, и, называя его контрактом, а не интерфейсом, мы можем лучше отразить суть.

И то и другое представляет собой практически одно и то же — соглашение, что класс предоставит определенные методы с конкретной сигнатурой.

Пространство имен `Illuminate\Contracts` содержит группу интерфейсов, которые реализуются компонентами Laravel с указанием их в подсказках типов. Это позволяет легко разрабатывать аналогичные компоненты, реализующие такие же интерфейсы, и использовать их в вашем приложении вместо стандартных компонентов `Illuminate`. Например, когда подсказки типов ядра и компонентов Laravel ссылаются на почтовый сервис, указываемым типом не является класс `Mailer`. Вместо этого используется ссылка на контракт (интерфейс) `Mailer`, что позволяет легко предоставить свой собственный почтовый сервис. Подробнее — в главе 11.

Контракт `Authenticatable` требует наличия методов (таких как `getAuthIdentifier()`), которые позволяли бы фреймворку аутентифицировать экземпляры данной модели в системе аутентификации. Трейт `Authenticatable` включает в себя методы, обеспечивающие выполнение этого контракта для типичной модели Eloquent.

Контракту `Authorizable` нужен метод типа `can()`, который позволял бы фреймворку авторизовать экземпляры данной модели с предоставлением им соответствующих

прав доступа в разных контекстах. Трейт `Authorizable` предоставляет методы, обеспечивающие выполнение контракта `Authorizable` для типичной модели Eloquent.

Наконец, контракт `CanResetPassword` требует методов вроде `getEmailForPasswordReset()` и `sendPasswordResetNotification()`, которые позволяли бы фреймворку сбрасывать пароль любой сущности, удовлетворяющей данному контракту. Трейт `CanResetPassword` предоставляет методы, обеспечивающие выполнение этого контракта для типичной модели Eloquent.

Мы уже можем легко представлять отдельного пользователя в базе данных (с помощью миграции) и извлекать его с помощью экземпляра модели, для которого мы можем выполнить аутентификацию (вход/выход в системе), авторизацию (проверку на наличие прав доступа к определенному ресурсу) и отправку электронного письма для сброса пароля.

Использование глобального хелпера `auth()` и фасада `Auth`

Использовать глобальный хелпер `auth()` — самый простой способ взаимодействия со статусом аутентифицированного пользователя в приложении. Получить ту же функциональность можно с помощью экземпляра класса `Illuminate\Auth\AuthManager` или фасада `Auth`.

Наиболее распространенный подход состоит в том, чтобы проверить, вошел ли пользователь в систему (метод `auth()->check()` возвращает значение `true`, если текущий пользователь вошел в систему; метод `auth()->guest()` возвращает значение `true`, если пользователь не вошел в систему), а затем получить имя текущего вошедшего в систему пользователя (используя метод `auth()->user()` или `auth()->id()`, если достаточно получить только идентификатор; оба метода возвращают значение `null` при отсутствии вошедших в систему пользователей).

В примере 9.3 показано, как можно использовать данный глобальный хелпер в контроллере.

Пример 9.3. Пример использования глобального хелпера `auth()` в контроллере

```
public function dashboard()
{
    if (auth()->guest()) {
        return redirect('sign-up');
    }

    return view('dashboard')
        ->with('user', auth()->user());
}
```

Далее мы рассмотрим внутреннее устройство системы аутентификации. Это полезная, но не *абсолютно необходимая* информация. Если хотите пока ее пропустить, переходите к разделу «Каркас аутентификации» на с. 244.

Контроллеры аутентификации

Разберемся, как именно осуществляется аутентификация пользователей и сброс пароля.

Все это происходит в нескольких контроллерах, расположенных в пространстве имен `Auth`: `RegisterController`, `LoginController`, `ResetPasswordController` и `ForgotPasswordController`.

Контроллер `RegisterController`

Контроллер `RegisterController` в сочетании с трейтом `RegistersUsers` содержит рационально выбранные значения по умолчанию, на основе которых будет выполняться отображение формы регистрации новым пользователям, валидация вводимых ими данных, создание новых пользователей после успешного выполнения валидации ввода и последующее перенаправление.

Сам контроллер содержит несколько хуков, которые вызываются трейтами в определенных точках. Это позволяет легко настраивать несколько типичных вариантов поведения без глубоких изменений в том коде, который заставляет все это работать.

Свойство `$redirectTo` определяет, куда будут перенаправляться пользователи после успешной регистрации. Метод `validator()` определяет способ валидации регистрационных данных. А метод `create()` — как создать нового пользователя на основе полученных регистрационных данных. Пример 9.4 демонстрирует контроллер `RegisterController` по умолчанию.

Пример 9.4. Контроллер `RegisterController`, предоставляемый фреймворком `Laravel` по умолчанию

```
...
class RegisterController extends Controller
{
    use RegistersUsers;

    protected $redirectTo = '/home';

    ...

    protected function validator(array $data)
    {
```

```

        return Validator::make($data, [
            'name' => 'required|string|max:255',
            'email' => 'required|string|email|max:255|unique:users',
            'password' => 'required|string|min:6|confirmed',
        ]);
    }

    protected function create(array $data)
    {
        return User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
        ]);
    }
}

```

Трейт RegistersUsers

Трейт `RegistersUsers`, импортируемый контроллером `RegisterController`, выполняет несколько основных функций процесса регистрации. Во-первых, он отображает пользователям представление формы регистрации с помощью метода `showRegistrationForm()`. Если хотите использовать для регистрации новых пользователей другое представление вместо `auth.register`, можно переопределить метод `showRegistrationForm()` в контроллере `RegisterController`.

Во-вторых, данный трейт обрабатывает `POST`-запрос, отправляемый из регистрационной формы, используя метод `register()`. Он передает введенные пользователем регистрационные данные для валидации в `validator()` контроллера `RegisterController`, а затем — в `create()`.

В-третьих, метод `redirectPath()` (подгружаемый посредством `RedirectsUsers`) определяет, куда должны перенаправляться пользователи после успешного прохождения регистрации. Можете определить этот URI-идентификатор, используя свойство `$redirectTo` контроллера или переопределив метод `redirectPath()` таким образом, чтобы он возвращал нужный вам путь.

Если нужно, чтобы данный трейт использовал другой гард (`guard`) аутентификации вместо предоставляемого по умолчанию (подробно рассказано в разделе «Гарды» на с. 250), можете переопределить метод `guard()` так, чтобы он возвращал нужный вам гард.

Контроллер LoginController

Контроллер `LoginController` позволяет пользователю выполнить вход в систему. Он подгружает трейт `AuthenticatesUsers`, который подгружает `RedirectsUsers` и `ThrottlesLogins`.

Как и контроллер `RegistrationController`, контроллер `LoginController` имеет свойство `$redirectTo` для задачи пути, по которому будет перенаправляться пользователь после успешного входа в систему. Все остальное содержит трейт `AuthenticatesUsers`.

Трейт `AuthenticatesUsers`

Трейт `AuthenticatesUsers` отвечает за отображение формы входа в систему, валидацию введенных данных, отклонение неверных данных, выход из системы и перенаправление пользователей после успешного входа.

Метод `showLoginForm()` по умолчанию отображает пользователю представление `auth.login`, но его можно переопределить так, чтобы он отображал другое представление.

Метод `login()` принимает POST-запрос формы входа в систему. Он выполняет валидацию запроса, используя метод `validateLogin()`, который можно переопределить для нестандартной валидации. Затем он подключается к функциональности трейта `ThrottlesLogins`, о котором мы поговорим чуть позже, для отклонения пользователей со слишком большим количеством неудачных попыток входа в систему. И он перенаправляет пользователя либо по указанному им пути (если пользователь был перенаправлен на страницу входа при попытке посетить одну из страниц приложения), либо по пути, определенному методом `redirectPath()`, который возвращает свойство `$redirectTo`.

После успешного входа в систему данный трейт вызывает пустой метод `authenticated()`. Поэтому, если вы хотите выполнять какое-либо поведение в ответ на успешный вход в систему, переопределите его в контроллере `LoginController`.

Метод `username()` определяет, какой из столбцов таблицы `users` следует использовать в качестве имени пользователя. По умолчанию это столбец `email`, но подойдет и другой столбец: переопределите метод `username()` контроллера таким образом, чтобы он возвращал имя нужного вам столбца.

Как и в случае трейта `RegistersUsers`, вы можете переопределить метод `guard()`, указав, какой гард аутентификации (подробнее — в разделе «Гарды» на с. 250) должен использовать данный контроллер.

Трейт `ThrottlesLogins`

Трейт `ThrottlesLogins` — это интерфейс для доступа к классу `Illuminate\Cache\RateLimiter` фреймворка Laravel для ограничения частоты запросов любого события с использованием кэша. Данный трейт применяет ограничение частоты запросов к процедуре входа в систему, запрещая использовать форму входа пользователям

с большим количеством неудачных попыток авторизации за определенный промежуток времени. Функции нет в Laravel 5.1.

При импорте трейта `ThrottlesLogins` имейте в виду, что все его методы являются защищенными (`protected`). Значит, к ним нельзя осуществлять доступ как к маршрутам. Вместо этого трейт `AuthenticatesUsers` проверяет, импортируете ли вы трейт `ThrottlesLogins`. Если да, наделяет этой функциональностью ваши формы для входа в систему без дополнительных усилий с вашей стороны. Поскольку по умолчанию контроллер `LoginController` импортирует эти трейты, вы легко получите данную функциональность, если решите использовать каркас аутентификации (см. раздел «Каркас аутентификации» на с. 244).

Трейт `ThrottlesLogins` позволяет выполнять не более пяти попыток в течение 60 секунд для любой комбинации имени пользователя и IP-адреса. За счет кэша он увеличивает счетчик неудачных входов в систему для заданной комбинации имени пользователя и IP-адреса. Если у какого-либо пользователя число неудачных попыток достигает пяти до истечения 60 секунд, он перенаправляется на страницу входа в систему с соответствующей ошибкой до истечения 60 секунд.

Контроллер `ResetPasswordController`

Контроллер `ResetPasswordController` подгружает трейт `ResetsPasswords`. Он обеспечивает валидацию и доступ к основным представлениям сброса пароля, а затем использует экземпляр класса `PasswordBroker` фреймворка Laravel (или любого другого класса, реализующего интерфейс `PasswordBroker`, если вы решите использовать собственный класс) для отправки электронного письма по поводу сброса пароля и для фактического сброса пароля.

Данный трейт отображает представление сброса пароля (метод `showResetForm()` отображает представление `auth.passwords.reset`) и обрабатывает POST-запрос, отправляемый из этого представления (метод `reset()` выполняет его валидацию и отправляет соответствующий ответ). Метод `resetPassword()` осуществляет непосредственный сброс пароля; при этом можно определить собственную реализацию брокера в `broker()` и гарда аутентификации в `guard()`.

Если вам нужно использовать собственный вариант какого-либо из этих поведений, переопределите соответствующий метод в контроллере.

Контроллер `ForgotPasswordController`

Контроллер `ForgotPasswordController` подгружает трейт `SendsPasswordResetEmails`. Он отображает форму `auth.passwords.email` с помощью метода `showLinkRequestForm()` и обрабатывает POST-запрос этой формы с помощью метода `sendResetLinkEmail()`. Можно определить собственную реализацию брокера в методе `broker()`.

Контроллер VerificationController

Контроллер `VerificationController` подгружает трейт `VerifiesEmails`, который проверяет адреса электронной почты вновь зарегистрированных пользователей. Вы можете задать собственный путь для перенаправления пользователей после успешного прохождения ими проверки.

Метод Auth::routes()

У нас уже есть контроллеры аутентификации, которые предоставляют некоторые методы для ряда predefined маршрутов, осталось позаботиться о том, чтобы наши пользователи могли *перейти* по этим маршрутам. Хотя мы можем добавить все маршруты вручную в файле `routes/web.php`, уже предусмотрено такое удобное средство, как метод `Auth::routes()`:

```
// routes/web.php
Auth::routes();
```

Метод `Auth::routes()` заносит набор predefined маршрутов в файл маршрутов. Пример 9.5 показывает, какие маршруты в действительности определяются в этом файле.

Пример 9.5. Маршруты, предоставляемые методом Auth::routes()

```
// Маршруты аутентификации
$this->get('login', 'Auth\LoginController@showLoginForm')->name('login');
$this->post('login', 'Auth\LoginController@login');
$this->post('logout', 'Auth\LoginController@logout')->name('logout');

// Маршруты регистрации
$this->get('register', 'Auth\RegisterController@showRegistrationForm')
    ->name('register');
$this->post('register', 'Auth\RegisterController@register');

// Маршруты сброса пароля
$this->get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')
    ->name('password.request');
$this->post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail')
    ->name('password.email');
$this->get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm')
    ->name('password.reset');
$this->post('password/reset', 'Auth\ResetPasswordController@reset');

// Если включена проверка адресов электронной почты
$this->get('email/verify', 'Auth\VerificationController@show')
    ->name('verification.notice');
$this->get('email/verify/{id}', 'Auth\VerificationController@verify')
    ->name('verification.verify');
$this->get('email/resend', 'Auth\VerificationController@resend')
    ->name('verification.resend');
```

Основную часть маршрутов метода `Auth::routes()` составляют маршруты для аутентификации, регистрации и сброса пароля. Имеются также и опциональные маршруты для проверки адресов электронной почты; данная возможность появилась в Laravel 5.7.

5.7 Чтобы включить сервис верификации адресов электронной почты Laravel, обеспечивающий подтверждение новыми пользователями наличия у них доступа к указанному при регистрации адресу электронной почты, перепишите вызов метода `Auth::routes()` следующим образом:

```
Auth::routes(['verify' => true]);
```

Мы обсудим это подробно далее, в разделе «Верификация адресов электронной почты» на с. 249.



В приложениях, использующих Laravel 5.7 или более новую версию, можно использовать метод `Auth::routes()`, отключив при этом ссылки для регистрации и/или сброса пароля. Нужно добавить соответствующие ключи, `register` и `reset`, в массив, передаваемый методу `Auth::routes()`:

```
Auth::routes(['register' => false, 'reset' => false]);
```

Каркас аутентификации

У вас уже есть миграция, модель, контроллеры и маршруты для системы аутентификации. Но что насчет представлений?

Фреймворк Laravel предоставляет *каркас* аутентификации (доступный с версии 5.2), который запускается в новом приложении и предоставляет вам дополнительный скелет кода, позволяющий ускорить ввод в действие системы аутентификации.

Каркас аутентификации в файл маршрутов добавляет маршруты метода `Auth::routes()`, представление для каждого маршрута и создает контроллер `HomeController`, выполняющий функции целевой страницы для зарегистрированных пользователей. При этом он определяет URI-идентификатор `/home` в качестве маршрута к методу `index()` контроллера `HomeController`.

Выполните команду `php artisan make:auth`, и вам станут доступны следующие файлы:

```
app/Http/Controllers/HomeController.php
resources/views/auth/login.blade.php
resources/views/auth/register.blade.php
resources/views/auth/verify.blade.php
resources/views/auth/passwords/email.blade.php
resources/views/auth/passwords/reset.blade.php
resources/views/layouts/app.blade.php
resources/views/home.blade.php
```

К этому моменту маршрут `/` возвращает представление страницы приветствия `welcome`, маршрут `/home` возвращает представление домашней страницы `home`, а маршруты аутентификации для входа в систему/выхода из нее, регистрации и сброса пароля указывают на контроллеры авторизации. Каждое из заполняемых начальными данными представлений имеет ориентированные на начальную загрузку макеты и поля формы для ввода всех необходимых данных при входе в систему, регистрации и сбросе пароля. И все это уже с правильными маршрутами.

Теперь вы располагаете необходимыми компонентами для каждого этапа стандартного процесса регистрации и аутентификации пользователей. Вы еще можете выполнить некоторые настройки и уже полностью готовы выполнять регистрацию и аутентификацию пользователей.

Представим вкратце последовательность шагов с момента создания нового сайта до получения полноценной системы аутентификации:

```
laravel new MyApp
cd MyApp
```

Указание в файле `.env` правильных параметров подключения к базе данных

```
php artisan make:auth
php artisan migrate
```

Вот и все! Запустите эти команды, и вы получите целевую страницу и ориентированную на начальную загрузку систему регистрации пользователей, входа, выхода и сброса пароля с простейшей целевой страницей для всех аутентифицированных пользователей.

Токен «Запомнить меня»

Хотя эта функциональность предлагается каркасом аутентификации «из коробки», будет полезно узнать, как она работает и как ее можно использовать самостоятельно. Если нужно реализовать долгосрочный токен доступа наподобие токена «Запомнить меня», убедитесь, что в вашей таблице `users` есть столбец `remember_token` (это будет так, если вы использовали миграцию по умолчанию).

Когда вы производите обычный вход пользователя в систему (например, применяя контроллер `LoginController` в сочетании с трейтом `AuthenticatesUsers`), вы пытаетесь выполнить аутентификацию с использованием предоставленных пользователем данных, как показано в примере 9.6.

Пример 9.6. Попытка выполнить аутентификацию пользователя

```
if (auth()->attempt([
    'email' => request()->input('email'),
    'password' => request()->input('password'),
])) {
    // Обработка успешного входа
}
```

При этом вы получаете регистрационные данные, которые сохраняются до завершения сессии пользователя. Если нужно, чтобы Laravel сохранял эти регистрационные данные неопределенно долго с помощью cookie-файлов (пока пользователь использует один и тот же компьютер и не выходит из системы), вы можете передать логическое значение `true` в качестве второго параметра метода `auth()->attempt()`. Пример 9.7 демонстрирует такой запрос.

Пример 9.7. Попытка выполнить аутентификацию пользователя с проверкой флажка «Запомнить меня»

```
if (auth()->attempt([
    'email' => request()->input('email'),
    'password' => request()->input('password'),
], request()->filled('remember'))) {
    // Обработка успешного входа
}
```

Мы проверяем, содержат ли введенные данные непустое свойство `remember`, которое возвращает логическое значение. Тем самым мы даем пользователям возможность указать, нужно ли их запоминать, используя флажок в форме входа в систему.

После этого можно вручную проверять, был ли текущий пользователь аутентифицирован токеном запоминания, используя метод `auth()->viaRemember()`, который возвращает логическое значение, указывающее, аутентифицирован ли текущий пользователь с помощью токена запоминания. Таким образом, вы можете сделать некоторые важные возможности недоступными для пользователей, аутентифицированных с помощью токена запоминания, требуя от них повторного введения пароля.

Выполнение вручную аутентификации пользователей

Наиболее типичный сценарий аутентификации пользователей сводится к следующему: вы даете пользователю возможность предоставить свои регистрационные данные, а затем с помощью метода `auth()->attempt()` проверяете соответствие предоставленных регистрационных данных кому-либо из реальных пользователей. Если это так, вы осуществляете вход пользователя в систему.

Иногда важна возможность осуществления аутентификации пользователя по своей инициативе. Например, дать администраторам право переключать пользователей.

Для этого существует четыре способа. Во-первых, можно просто передавать идентификатор пользователя:

```
auth()->loginUsingId(5);
```

Во-вторых, можно передавать объект `User` (или любой другой объект, реализующий контракт `Illuminate\Contracts\Auth\Authenticatable`):

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

Третий и четвертый способы состоят в том, чтобы методом `once()` или `onceUsingId()` аутентифицировать определенного пользователя только для текущего запроса без какого-либо влияния на сессию или cookie-файлы:

```
auth()->once(['username' => 'mattstauffer']);  
// или  
auth()->onceUsingId(5);
```

Обратите внимание, что передаваемый методу `once()` массив может содержать любые пары «ключ/значение», позволяющие однозначно идентифицировать аутентифицируемого пользователя. Вы даже можете передавать несколько ключей и значений, если это целесообразно для вашего проекта. Например:

```
auth()->once([  
    'last_name' => 'Stauffer',  
    'zip_code' => 90210,  
])
```

Выполнение вручную выхода пользователя из системы

Чтобы вручную выполнить выход пользователя из системы, вызовите метод `logout()`:

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

5.6 Аннулирование сессий на других устройствах. Если нужно выполнять выход не только из текущей сессии пользователя, но и из сессий на любых других устройствах, то потребуется запрашивать у пользователя пароль и передавать его методу `logoutOtherDevices()` (доступный в Laravel 5.6 и более новых версиях). Для этого нужно добавить middleware-метод `AuthenticateSession` (закомментированный по умолчанию) в группу `web` в файле `app\Http\Kernel.php`:

```
'web' => [  
    // ...  
    \Illuminate\Session\Middleware\AuthenticateSession::class,  
],
```

После этого можно использовать данный метод в нужном месте во встроенном коде:

```
auth()->logoutOtherDevices($password);
```

auth

Пример 9.3 демонстрирует, как можно проверять, аутентифицирован ли посетитель, и перенаправлять его, если это не так. Вы можете выполнять такие проверки для каждого маршрута приложения, однако это очень быстро потребует большой затраты сил. В то же время с задачей предоставления доступа к определенным маршрутам только для гостей или только для аутентифицированных пользователей прекрасно справляется маршрутное middleware (о том, как оно работает, написано в главе 10).

Laravel предоставляет нужное нам middleware, что называется, «из коробки». Эти маршрутное middleware определено в классе `App\Http\Kernel`:

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

С аутентификацией связаны четыре из предоставляемых по умолчанию маршрутных middleware-методов.

- ☐ `auth` предоставляет доступ к маршруту только аутентифицированным пользователям.
- ☐ `auth.basic` — доступ к маршруту только для аутентифицированных пользователей с помощью базовой HTTP-аутентификации.
- ☐ `guest` обеспечивает доступ только для неаутентифицированных пользователей.
- ☐ `can` дает пользователям доступ только к определенным маршрутам.

Наиболее часто используются `auth` — для тех разделов приложения, которые должны быть доступными только аутентифицированным пользователям, и `guest` — для разделов, которые не нужно отображать для аутентифицированных пользователей (форма входа в систему). `auth.basic` применяется гораздо реже, для аутентификации с использованием заголовков запросов.

Пример 9.8 демонстрирует, как можно ограничить доступ к нескольким маршрутам с помощью `auth`.

Пример 9.8. Пример ограничения доступа к маршрутам с помощью `auth`

```
Route::middleware('auth')->group(function () {
    Route::get('account', 'AccountController@dashboard');
});
```

```
Route::get('login', 'Auth/LoginController@getLogin')->middleware('guest');
```


Верификация адресов электронной почты

5.7 В Laravel 5.7 появилась новая возможность, позволяющая проверять наличие у пользователей доступа к указанному при регистрации адресу электронной почты.

Чтобы активировать верификацию адресов электронной почты, отредактируйте класс `App\User` так, чтобы он реализовывал контракт `Illuminate\Contracts\Auth\MustVerifyEmail`, как показано в примере 9.9.

Пример 9.9. Добавление трейта `MustVerifyEmail` в модель `Authenticatable`

```
class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

Таблица `users` также должна содержать обнуляемый столбец для временных меток с именем `email_verified_at`. Если вы используете Laravel 5.7 или более новую версию, то такой столбец будет создан по умолчанию новой миграцией `CreateUsersTable`.

Наконец, необходимо активировать маршруты верификации адресов электронной почты в контроллере. Самый простой способ — использовать метод `Auth::routes()` в файле маршрутов с параметром `verify`, установленным в `true`:

```
Auth::routes(['verify' => true]);
```

Теперь можно закрыть доступ к любому маршруту для тех пользователей, которые не подтвердили адрес электронной почты:

```
Route::get('posts/create', function () {
    // Доступно только для проверенных пользователей...
})->middleware('verified');
```

При этом можно указать собственный маршрут для перенаправления пользователей после проверки их контроллером `VerificationController`:

```
protected $redirectTo = '/profile';
```

Blade-директивы для аутентификации

Если вам нужно проверять аутентификацию пользователя на уровне представлений, а не маршрута, помогут директивы `@auth` и `@guest` (пример 9.10).

Пример 9.10. Проверка статуса аутентификации пользователя в шаблонах

```
@auth
    // Пользователь аутентифицирован
@endauth

@guest
    // Пользователь не аутентифицирован
@endguest
```

Можно также указать, какой гард должен использоваться с этими директивами, передав имя гарда в качестве параметра, как показано в примере 9.11.

Пример 9.11. Проверка статуса аутентификации в шаблонах с использованием конкретного гарда аутентификации

```
@auth('trainees')  
    // Пользователь аутентифицирован  
@endauth  
  
@guest('trainees')  
    // Пользователь не аутентифицирован  
@endguest
```

Гарды

Маршрутизация всех аспектов системы аутентификации фреймворка Laravel осуществляется с использованием так называемых *гардов* (*guard*). Каждый гард состоит из двух элементов: *драйвера* (например, с именем `session`), который определяет способ сохранения и извлечения статуса аутентификации, и *провайдера* (скажем, с именем `users`), который обеспечивает отбор пользователей по определенным критериям.

В Laravel по умолчанию входят два гарда: `web` и `api`. Гард `web` обеспечивает более традиционный способ аутентификации: драйвером `session` и базовым провайдером пользователей. Гард `api` применяет тот же провайдер пользователей, но вместо `session` использует драйвер `token` для аутентификации каждого запроса.

Вы можете изменить драйверы, если требуется по-разному идентифицировать пользователей и сохранять их идентификационные данные (например, переключаться от использования продолжительной сессии к предоставлению токена при каждой загрузке страницы), и изменить провайдеры, если нужно варьировать способ сохранения или извлечения пользователей (скажем, сохранять некоторых пользователей с помощью Mongo, а не MySQL).

Указание другого гарда по умолчанию

Гарды определены в файле `config/auth.php`. Вы можете указать другие гарды, добавить новые, а также указать гард по умолчанию. Следует отметить, что это нужно редко. В большинстве приложений, создаваемых с помощью Laravel, применяется лишь один гард.

Гард по умолчанию используется в том случае, когда какие-либо возможности аутентификации применяются без указания гарда. Например, вызов метода `auth()->user()` приведет к извлечению текущего аутентифицированного пользо-

вателя гардом по умолчанию. Чтобы задать другой гард по умолчанию, измените значение, присваиваемое параметру `auth.defaults.guard` в файле `config/auth.php`:

```
'defaults' => [
    'guard' => 'web', // Измените указанное здесь значение по умолчанию
    'passwords' => 'users',
],
```

При использовании Laravel 5.1 структура аутентификационных данных будет выглядеть немного иначе. Однако не стоит сильно волноваться — все возможности ведут себя точно так же; они лишь немного иначе организованы.



Соглашения в отношении конфигурации

Упомянув в тексте разделы конфигурации, я использую выражения вида `auth.defaults.guard`. Оно обозначает следующее: в файле `config/auth.php` в разделе массива с ключом `defaults` должно присутствовать свойство с ключом `guard`.

Использование других гардов без изменения базового

При необходимости можно использовать другой гард, *не указывая* его в качестве гарда по умолчанию. Для этого следует вызывать метод `auth()` вместе с методом `guard()`:

```
$apiUser = auth()->guard('api')->user();
```

Данный код обеспечит только для этого вызова извлечение текущего пользователя с использованием гарда `api`.

Добавление нового гарда

Вы в любой момент можете добавить новый гард в файле `config/auth.php`, изменив определение раздела `auth.guards`:

```
'guards' => [
    'trainees' => [
        'driver' => 'session',
        'provider' => 'trainees',
    ],
],
```

Здесь мы создаем новый гард с именем `trainees` (в дополнение к `web` и `api`). Допустим, нужно создать приложение, пользователями которого будут спортивные тренеры со своими пользователями: спортсменами, которые смогут входить в свои поддомены. Таким образом, нужен отдельный гард для второй категории пользователей.

Свойству `driver` можно присвоить одно из двух значений: `token` и `session`. Что касается свойства `provider`, то «из коробки» доступно только значение `users`, которое обеспечивает аутентификацию на основе используемой по умолчанию таблицы `users`. Однако вы можете легко создать и собственного провайдера.

Гарды на основе замыкания запроса

Если нужно определить собственный гард, при этом условия (определяющие способ поиска пользователей на основе запроса) можно достаточно просто описать в ответе на любой заданный HTTP-запрос, то вы можете поместить код поиска пользователей в замыкание, не обременяя себя созданием нового пользовательского класса гарда.

Метод аутентификации `viaRequest()` позволяет определить гард (с именем, переданным в качестве первого параметра), используя только замыкание (определенное во втором параметре), которое принимает HTTP-запрос и возвращает имя соответствующего пользователя. Чтобы зарегистрировать такой гард на основе замыкания запроса, следует вызвать метод `viaRequest()` внутри метода `boot()` класса `AuthServiceProvider`, как показано в примере 9.12.

Пример 9.12. Определение гарда на основе замыкания запроса

```
public function boot()
{
    $this->registerPolicies();

    Auth::viaRequest('token-hash', function ($request) {
        return User::where('token-hash', $request->token)->first();
    });
}
```

Создание собственного провайдера пользователей

Доступные провайдеры определяются в файле `config/auth.php`, в разделе `auth.providers`, расположенном непосредственно под определением гардов. Создадим нового провайдера с именем `trainees`:

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\User::class,
    ],

    'trainees' => [
        'driver' => 'eloquent',
        'model' => App\Trainee::class,
    ],
],
```

Свойству `driver` можно присвоить одно из двух значений: `eloquent` и `database`. При выборе `eloquent` потребуется свойство `model`, содержащее имя класса `Eloquent` (то есть модели, используемой для вашего класса `User`); а при `database` нужно свойство `table`, указывающее, на основе какой таблицы будет производиться аутентификация.

В данном приложении используется две модели: `User` и `Trainee`. Они должны аутентифицироваться по отдельности. Это позволяет коду различать пользователей (тренеров) — `auth()->guard('users')` и спортсменов — `auth()->guard('trainees')`.

Еще одно замечание: маршрутный `middleware`-метод `auth` может принимать имя гарда в качестве параметра. Это позволяет ограничить доступ к определенным маршрутам, используя конкретный гард:

```
Route::middleware('auth:trainees')->group(function () {  
    // Маршруты, доступные только для спортсменов  
});
```

Собственные провайдеры пользователей для нереляционных баз данных

Описанный выше способ создания собственного провайдера пользователей по-прежнему опирается на использование класса `UserProvider`, что подразумевает извлечение идентификационных данных из реляционной БД. Однако при использовании `Mongo`, `Riak` или другой аналогичной СУБД вы не сможете обойтись без создания собственного класса.

Для этого создайте новый класс, который реализует интерфейс `Illuminate\Contracts\Auth\UserProvider`, а затем подключите его в методе `AuthServiceProvider@boot`:

```
auth()->provider('riak', function ($app, array $config) {  
    // Возвращает экземпляр класса Illuminate\Contracts\Auth\UserProvider...  
    return new RiakUserProvider($app['riak.connection']);  
});
```

События аутентификации

Мы подробно поговорим о событиях в главе 16. Пока отмечу, что система событий `Laravel` представляет собой простейшую платформу публикации/подписки. Она обеспечивает оповещение о системных и пользовательских событиях, и пользователь может создавать слушатели событий, выполняющие конкретные действия в ответ на определенные события.

Например, как обеспечить отправку пинга конкретному сервису безопасности при каждой блокировке пользователя из-за превышения лимита на количество

неудачных попыток входа в систему? Такой сервис может устанавливать лимит на количество неудачных входов для определенных географических регионов или производить другие действия. Конечно, вы можете встроить вызов в соответствующем контроллере. Однако если воспользоваться событиями, достаточно создать и зарегистрировать слушатель для события «блокировка пользователя».

Полный список событий, генерируемых системой аутентификации, показан в примере 9.13.

Пример 9.13. Генерируемые фреймворком события аутентификации

```
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [],
    'Illuminate\Auth\Events\Attempting' => [],
    'Illuminate\Auth\Events\Authenticated' => [],
    'Illuminate\Auth\Events>Login' => [],
    'Illuminate\Auth\Events\Failed' => [],
    'Illuminate\Auth\Events\Logout' => [],
    'Illuminate\Auth\Events\Lockout' => [],
    'Illuminate\Auth\Events>PasswordReset' => [],
];
```

Система аутентификации генерирует события «регистрация выполнена» (**Registered**), «попытка входа в систему» (**Attempting**), «аутентификация выполнена» (**Authenticated**), «успешный вход в систему» (**Login**), «неудачный вход в систему» (**Failed**), «выход из системы» (**Logout**), «блокировка» (**Lockout**) и «сброс пароля» (**PasswordReset**). О том, как создать слушатели для этих событий, написано в главе 16.

Система авторизации (список управления доступом) и роли

5.2 Рассмотрим систему авторизации Laravel. Она позволяет проверять, *авторизован* ли пользователь для выполнения определенных действий, используя следующие базовые команды: **can**, **cannot**, **allows** и **denies**. Такая система авторизации на основе списка управления доступом (access control list, ACL) была впервые введена в Laravel 5.2.

Данное управление авторизацией осуществляется главным образом за счет фасада **Gate**, однако можно использовать удобные хелперы в своих контроллерах и в модели **User** в виде **middleware** и **Blade-директив**. Определенное представление о возможностях в примере 9.14.

Пример 9.14. Простейший пример использования фасада **Gate**

```
if (Gate::denies('edit-contact', $contact)) {
    abort(403);
}
```

```
if (! Gate::allows('create-contact', Contact::class)) {  
    abort(403);  
}
```

Определение правил авторизации

По умолчанию правила авторизации определяются в методе `boot()` класса `AuthServiceProvider`, где вызываются методы фасада `Auth`.

Правило авторизации называется *способностью* и состоит из двух элементов: строкового ключа (например, `update-contact`) и замыкания, возвращающего логическое значение. Пример 9.15 демонстрирует способность обновления контакта.

Пример 9.15. Пример способности обновления контакта

```
class AuthServiceProvider extends ServiceProvider  
{  
    public function boot()  
    {  
        $this->registerPolicies();  
  
        Gate::define('update-contact', function ($user, $contact) {  
            return $user->id == $contact->user_id;  
        });  
    }  
}
```

Пошагово рассмотрим процесс определения способности.

Первое, что вы должны сделать, — это определить ключ. В качестве имени ключа желательно выбрать строку, отражающую суть предоставляемой пользователю способности в контексте вашего кода. В приведенных выше примерах кода имена соответствуют принятому в Laravel формату `{команда}-{имяМодели}`: `create-contact`, `update-contact` и т. д.

Второй шаг — определение замыкания. Как первый параметр следует указать текущего аутентифицированного пользователя, а в качестве остальных параметров — те объекты, наличие доступа к которым вы проверяете, — в данном случае это контакт.

Получив эти два объекта, можно проверить, авторизован ли пользователь для обновления указанного контакта. Вы можете задать здесь любую логику, но в данном случае результат авторизации определяется тем, является ли пользователь создателем строки контакта. Это замыкание вернет значение `true` (пользователь авторизован), если контакт был создан текущим пользователем, и значение `false` (пользователь не авторизован) — в противном случае.

Как и в случае с определениями маршрутов, для разрешения этого определения вместо замыкания можно использовать класс и метод:

```
$gate->define('update-contact', 'ContactACLChecker@updateContact');
```

Фасад Gate (и его внедрение)

Теперь, когда у вас уже определена способность, можно выполнить проверку на ее основе. Самый простой способ — использовать фасад Gate, как показано в примере 9.16 (или внедрить экземпляр класса `Illuminate\Contracts\Auth\Access\Gate`).

Пример 9.16. Простейший способ использования фасада Gate

```
if (Gate::allows('update-contact', $contact)) {
    // Обновление контакта
}

// или
if (Gate::denies('update-contact', $contact)) {
    abort(403);
}
```

Можно определить способность с несколькими параметрами — например, в случае, когда контакты разбиты на группы и нужно проверять, может ли пользователь добавлять контакты в ту или иную группу. Как это сделать, показано в примере 9.17.

Пример 9.17. Способности с несколькими параметрами

```
// Определение
Gate::define('add-contact-to-group', function ($user, $contact, $group) {
    return $user->id == $contact->user_id && $user->id == $group->user_id;
});

// Использование
if (Gate::denies('add-contact-to-group', [$contact, $group])) {
    abort(403);
}
```

Если нужно проверить авторизацию какого-либо другого пользователя, помимо текущего аутентифицированного пользователя, используйте метод `forUser()`, как показано в примере 9.18.

Пример 9.18. Указание пользователя для фасада Gate

```
if (Gate::forUser($user)->denies('create-contact')) {
    abort(403);
}
```

Ресурсы гейтов

Наиболее частой областью использования списков управления доступом является определение доступа к отдельным «ресурсам» (например, модели Eloquent или иным сущностям, которыми могут распоряжаться пользователи с панели администратора).

Метод `resource()` позволяет применить к одному ресурсу сразу четыре часто используемых гейта — `view` (просмотр), `create` (создание), `update` (обновление) и `delete` (удаление).

```
Gate::resource('photos', 'App\Policies\PhotoPolicy');
```

Данный код эквивалентен следующим четырем строкам:

```
Gate::define('photos.view', 'App\Policies\PhotoPolicy@view');
Gate::define('photos.create', 'App\Policies\PhotoPolicy@create');
Gate::define('photos.update', 'App\Policies\PhotoPolicy@update');
Gate::define('photos.delete', 'App\Policies\PhotoPolicy@delete');
```

Authorize

Авторизовать целые маршруты можно с помощью `middleware`-метода `Authorize` (у которого есть сокращенный псевдоним `can`), как показано в примере 9.19.

Пример 9.19. Использование `Authorize`

```
Route::get('people/create', function () {
    // Создание пользователя
})->middleware('can:create-person');

Route::get('people/{person}/edit', function () {
    // Редактирование пользователя
})->middleware('can:edit,person');
```

Здесь параметр `{person}` (определяемый в виде строки или привязки модели маршрута) передается методу способности в качестве дополнительного параметра.

Первая проверка в примере 9.19 выполняется на основе обычной способности, а вторая — на основе политики. Мы подробно поговорим об этом в подразделе «Политики» на с. 261.

Если нужно проверить действие, которому не требуется экземпляр модели (например, действию `create` (создание), в отличие от действия `edit` (редактирование), не нужно передавать реальный экземпляр привязки модели маршрута), можно передать только имя класса:

```
Route::post('people', function () {
    // Создание пользователя
})->middleware('can:create,App\Person');
```

Авторизация внутри контроллера

Родительский класс `App\Http\Controllers\Controller` в Laravel импортирует трейт `AuthorizesRequests`, который предоставляет три метода авторизации: `authorize()`, `authorizeForUser()` и `authorizeResource()`.

`authorize()` принимает в качестве параметров ключ способности и объект (или массив объектов). А в случае неудачного результата авторизации выполняет выход из приложения с кодом состояния 403 (`Unauthorized` — «Не авторизован»). Это позволяет сократить три строки кода авторизации до всего одной строки, как показано в примере 9.20.

Пример 9.20. Упрощение авторизации внутри контроллера с помощью метода `authorize()`

```
// Исходный вариант:
public function edit(Contact $contact)
{
    if (Gate::cannot('update-contact', $contact)) {
        abort(403);
    }

    return view('contacts.edit', ['contact' => $contact]);
}

// Упрощенный вариант:
public function edit(Contact $contact)
{
    $this->authorize('update-contact', $contact);

    return view('contacts.edit', ['contact' => $contact]);
}
```

Метод `authorizeForUser()` отличается лишь тем, что позволяет передавать объект `User` вместо используемого по умолчанию текущего аутентифицированного пользователя:

```
$this->authorizeForUser($user, 'update-contact', $contact);
```

Метод `authorizeResource()`, вызываемый только один раз в конструкторе контроллера, сопоставляет predetermined набор правил авторизации с каждым методом RESTful-контроллера — примерно так, как в примере 9.21.

Пример 9.21. Сопоставление правил авторизации с методами с помощью метода `authorizeResource()`

```
...
class ContactsController extends Controller
{
    public function __construct()
    {
        // Данный вызов обеспечивает выполнение всех действий,
        // выполняемых внутри размещенных ниже методов.
        // Если вы поместите его здесь, можно будет удалить все вызовы
        // метода authorize() в размещенных здесь методах для отдельных ресурсов.
        $this->authorizeResource(Contact::class);
    }

    public function index()
    {

```

```
        $this->authorize('view', Contact::class);
    }

    public function create()
    {
        $this->authorize('create', Contact::class);
    }

    public function store(Request $request)
    {
        $this->authorize('create', Contact::class);
    }

    public function show(Contact $contact)
    {
        $this->authorize('view', $contact);
    }

    public function edit(Contact $contact)
    {
        $this->authorize('update', $contact);
    }

    public function update(Request $request, Contact $contact)
    {
        $this->authorize('update', $contact);
    }

    public function destroy(Contact $contact)
    {
        $this->authorize('delete', $contact);
    }
}
```

Проверка с помощью экземпляра класса User

За пределами контроллера обычно требуется проверять способности некоторого конкретного пользователя, а не текущего аутентифицированного. Мы уже умеем это делать, используя метод `forUser()` фасада `Gate`, однако это не всегда удобно.

Трейт `Authorizable` в классе `User` предоставляет три метода для определения более читаемого кода авторизации: `$user->can()`, `$user->cant()` и `$user->cannot()`. Методы `cant()` и `cannot()` аналогичны, а `can()` — их противоположность.

Это позволяет вам выполнять авторизацию так, как в примере 9.22.

Пример 9.22. Проверка на предмет авторизации с помощью экземпляра класса `User`

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

if ($user->can('create-contact')) {
    // Выполнение некоторых действий
}
```

Данные методы просто передают ваши параметры фасаду Gate. Так, в примере выше они обеспечат следующий вызов: `Gate::forUser($user)->check('create-contact')`.

Проверки с помощью Blade-директив

Можно применить удобный хелпер в виде Blade-директивы `@can`. Как использовать эту директиву, показано в примере 9.23.

Пример 9.23. Использование Blade-директивы `@can`

```
<nav>
    <a href="/">Home</a>
    @can('edit-contact', $contact)
        <a href="{ {{ route('contacts.edit', [$contact->id]) }}">Edit This Contact</a>
    @endcan
</nav>
```

Кроме того, можете использовать директиву `@else` между `@can` и `@endcan`, а также `@cannot` и `@endcannot`, как показано в примере 9.24.

Пример 9.24. Использование Blade-директивы `@cannot`

```
<h1>{{ $contact->name }}</h1>
@cannot('edit-contact', $contact)
    LOCKED
@endcannot
```

Перехват проверок

Если вам приходилось разрабатывать приложение с классом пользователей-администраторов, то после рассмотрения приведенных простейших примеров замыканий авторизации у вас, вероятно, возник вопрос: каким образом можно добавить класс суперпользователя, который будет каждый раз переопределять эти проверки? К счастью, у нас уже есть инструмент для этого.

В классе `AuthServiceProvider`, где вы уже определили свои способности, можно добавить вызов метода `before()`. Эта проверка будет выполняться до всех остальных тестов и при необходимости переопределять их, как показано в примере 9.25.

Пример 9.25. Переопределение проверок фасада Gate с помощью метода `before()`

```
Gate::before(function ($user, $ability) {
    if ($user->isOwner()) {
        return true;
    }
});
```

В числе прочего методу передается строковое имя способности, что позволяет различать используемые хуки `before()` по именам способностей.

Политики

Для всех рассмотренных выше элементов управления доступом вам требовалось вручную связывать модели Eloquent с именами способностей. Вы могли создать способность с именем наподобие `visit-dashboard`, которая не была бы связана с конкретной моделью Eloquent, но в большинстве рассмотренных примеров *производились конкретные действия в отношении определенной сущности* — и, как правило, этой *сущностью* была модель Eloquent.

Политики авторизации представляют собой организационные структуры, которые помогают группировать логику авторизации по тем ресурсам, доступ к которым вы контролируете. Они облегчают определение правил авторизации для поведений, относящихся к конкретной модели Eloquent (или к другому классу PHP) за счет размещения их в одном месте.

Генерирование политик

Политики — это классы PHP, которые можно сгенерировать с помощью следующей команды Artisan:

```
php artisan make:policy ContactPolicy
```

После генерирования политик их необходимо зарегистрировать. Для этой цели используется свойство `$policies` класса `AuthServiceProvider`, представляющее собой массив. Ключ каждого элемента этого массива — имя класса контролируемого ресурса (в большинстве случаев это класс Eloquent), а значение — имя класса политики. Как это выглядит, показывает пример 9.26.

Пример 9.26. Регистрация политик в классе `AuthServiceProvider`

```
class AuthServiceProvider extends ServiceProvider
{
    protected $policies = [
        Contact::class => ContactPolicy::class,
    ];
}
```

Класс политики, генерируемый командой Artisan, не имеет специальных свойств или методов, но каждый добавляемый вами метод будет сопоставляться как ключ способности данного объекта.



Автоопределение политик

В приложениях, использующих Laravel 5.8 или более новую версию, фреймворк Laravel автоматически устанавливает связи между политиками и соответствующими моделями. Например, он автоматически применит политику `PostPolicy` к модели `Post`.

Если нужно модифицировать логику, используемую Laravel для такого автоматического сопоставления, ознакомьтесь с документацией по политикам (<http://bit.ly/2HJ4itY>).

Чтобы посмотреть, как это работает, определим метод `update()` (пример 9.27).

Пример 9.27. Пример метода политики `update()`

```
<?php

namespace App\Policies;

class ContactPolicy
{
    public function update($user, $contact)
    {
        return $user->id == $contact->user_id;
    }
}
```

Обратите внимание, что содержимое этого метода выглядит так же, как в определении фасада `Gate`.



Методы политик, не использующие экземпляры

53 Что делать, когда требуется определить метод политики, относящийся к классу, но не к конкретному экземпляру, то есть отвечающий на вопрос вида «Может ли данный пользователь создавать контакты вообще?», а не «Может ли данный пользователь просматривать этот конкретный контакт?» В Laravel 5.3 такой метод можно использовать точно так же, как обычный метод. В Laravel 5.2 при создании такого метода к его имени нужно добавить окончание `Any` (Любой):

```
...
class ContactPolicy
{
    public function createAny($user)
    {
        return $user->canCreateContacts();
    }
}
```

Проверка политик

Если для конкретного типа ресурса задана некоторая политика, фасад `Gate` будет определять по первому параметру, какой метод политики следует проверять. Так, вызов `Gate::allows('update', $contact)` приведет к проверке на предмет авторизации метода `ContactPolicy@update`.

Это справедливо и для `Authorize`, а также проверок с помощью модели `User` и Blade-директив, как показывает пример 9.28.

Пример 9.28. Проверка на предмет авторизации на основе политики

```
// Gate
if (Gate::denies('update', $contact)) {
    abort(403);
}
```

```
// Gate при отсутствии явного экземпляра
if (! Gate::check('create', Contact::class)) {
    abort(403);
}

// User
if ($user->can('update', $contact)) {
    // Выполнение определенных действий
}

// Blade
@can('update', $contact)
    // Выполнение определенных действий
@endcan
```

Можно воспользоваться хелпером `policy()`, который позволяет извлечь класс политики и выполнить его методы:

```
if (policy($contact)->update($user, $contact)) {
    // Выполнение определенных действий
}
```

Переопределение политик

Как и в случае обычных способностей, в политиках можно определять метод `before()`, позволяющий переопределить любой вызов еще до его обработки (пример 9.29).

Пример 9.29. Переопределение политик с помощью метода `before()`

```
public function before($user, $ability)
{
    if ($user->isAdmin()) {
        return true;
    }
}
```

Тестирование

В тестах приложения часто требуется выполнять определенное поведение от имени конкретного пользователя. Поэтому необходима возможность аутентификации в качестве пользователя, а также проверки правил авторизации и маршрутов аутентификации.

Хотя вы можете написать тест приложения, который будет вручную выполнять переход на страницу входа в систему, а затем заполнять и отправлять форму, в этом нет необходимости. Лучше воспользуйтесь самым простым способом — симулируйте вход пользователя в систему с помощью метода `->be()`. Как это можно сделать, показано в примере 9.30.

Пример 9.30. Аутентификация пользователя в тестах приложения

```
public function test_it_creates_a_new_contact()
{
    $user = factory(User::class)->create();
    $this->be($user);

    $this->post('contacts', [
        'email' => 'my@email.com',
    ]);

    $this->assertDatabaseHas('contacts', [
        'email' => 'my@email.com',
        'user_id' => $user->id,
    ]);
}
```

Вместо метода `be()` также можно использовать, в том числе в цепочке методов, метод `activeAs()`, если он кажется более читабельным:

```
public function test_it_creates_a_new_contact()
{
    $user = factory(User::class)->create();

    $this->actingAs($user)->post('contacts', [
        'email' => 'my@email.com',
    ]);

    $this->assertDatabaseHas('contacts', [
        'email' => 'my@email.com',
        'user_id' => $user->id,
    ]);
}
```

Правила авторизации можно проверять так, как показано в примере 9.31.

Пример 9.31. Тестирование правил авторизации

```
public function test_non_admins_cant_create_users()
{
    $user = factory(User::class)->create([
        'admin' => false,
    ]);
    $this->be($user);

    $this->post('users', ['email' => 'my@email.com']);

    $this->assertDatabaseMissing('users', [
        'email' => 'my@email.com',
    ]);
}
```

Еще один способ — выполнять проверку на наличие ответа 403, как показано в примере 9.32.

Пример 9.32. Тестирование правил авторизации путем проверки кода состояния

```
public function test_non_admins_cant_create_users()
{
    $user = factory(User::class)->create([
        'admin' => false,
    ]);
    $this->be($user);

    $response = $this->post('users', ['email' => 'my@email.com']);

    $response->assertStatus(403);
}
```

Нужно также проверить работоспособность наших маршрутов аутентификации (регистрации и входа в систему), как показано в примере 9.33.

Пример 9.33. Тестирование маршрутов аутентификации

```
public function test_users_can_register()
{
    $this->post('register', [
        'name' => 'Sal Leibowitz',
        'email' => 'sal@leibs.net',
        'password' => 'abcdefg123',
        'password_confirmation' => 'abcdefg123',
    ]);

    $this->assertDatabaseHas('users', [
        'name' => 'Sal Leibowitz',
        'email' => 'sal@leibs.net',
    ]);
}

public function test_users_can_log_in()
{
    $user = factory(User::class)->create([
        'password' => Hash::make('abcdefg123')
    ]);

    $this->post('login', [
        'email' => $user->email,
        'password' => 'abcdefg123',
    ]);

    $this->assertTrue(auth()->check());
    $this->assertTrue($user->is(auth()->user()));
}
```

Для проверки всего процесса аутентификации можно воспользоваться возможностями интеграционного тестирования и симитировать в тесте щелчки кнопкой мыши на полях аутентификации и их отправку. Мы поговорим об этом подробнее в главе 12.



Различия в наименовании методов тестирования в версиях, предшествующих Laravel 5.4

В проектах, использующих версии до Laravel 5.4, метод `assertDatabaseHas()` следует заменить методом `seeInDatabase()`, а `assertDatabaseMissing()` — методом `dontSeeInDatabase()`. Кроме того, нужно вызывать метод `assertStatus()` объекта `$this`, а не объекта `$response`.

Резюме

Помимо модели `User` по умолчанию, миграции `create_users_table`, контроллеров и каркаса аутентификации, в Laravel по умолчанию входит полнофункциональная система аутентификации пользователей. При этом контроллер `RegisterController` отвечает за регистрацию пользователей, `LoginController` — за аутентификацию пользователей, а `ResetPasswordController` и `ForgotPasswordController` — за сброс пароля. Каждый из контроллеров имеет ряд свойств и методов, которые можно использовать для переопределения некоторых видов поведения по умолчанию.

Фасад `Auth` и глобальный хелпер `auth()` обеспечивают доступ к текущему пользователю (`auth()->user()`) и позволяют легко проверить, был ли выполнен вход в систему (`auth()->check()` и `auth()->guest()`).

Laravel также имеет встроенную систему авторизации, которая позволяет установить конкретные способности (`create-contact`, `visit-secret-page`) или политики, определяющие взаимодействие пользователя с моделью в целом.

Проверку на предмет авторизации можно выполнять с помощью фасада `Gate`, методов `can()` и `cannot()` класса `User`, Blade-директив `@can` и `@cannot`, метода `authorize()` контроллера, а также middleware-метода `can`.

10 Запросы, ответы и middleware

Я уже упоминал объект `Illuminate Request`. Так, в главе 3 показано, как можно указывать этот тип подсказки в конструкторах для получения экземпляра и извлекать его с помощью хелпера `request()`, а в главе 7 рассмотрено его использование для получения информации о вводимых пользователем данных.

В этой главе мы подробно поговорим о том, что собой представляет и как генерируется объект `Request`, о его смысле и роли в жизненном цикле вашего приложения. А также об объекте `Response` и о том, как в Laravel реализован шаблон `middleware`.

Жизненный цикл запроса в Laravel

Каждый запрос, поступающий в приложение Laravel, вне зависимости от того, был ли он сгенерирован HTTP-запросом или командой интерфейса, немедленно преобразуется в объект `Illuminate Request`, который, пройдя через ряд промежуточных уровней, в итоге подвергается синтаксическому разбору самим приложением. После этого приложение генерирует объект `Illuminate Response`, который отправляется обратно через все эти слои и возвращается конечному пользователю.

Этот жизненный цикл запроса/ответа показан на рис. 10.1. Посмотрим, каким образом реализуется каждый из этих шагов, начиная с первой строки кода и заканчивая последней.

Начальная загрузка приложения

Каждое приложение Laravel обладает конфигурацией, которая настроена на уровне веб-сервера — в файле Apache `.htaccess`, в параметре конфигурации Nginx или в чем-то аналогичном — и захватывает каждый веб-запрос независимо от URL-адреса, направляя его к файлу `public/index.php` в каталоге приложения Laravel (app).



Рис. 10.1. Жизненный цикл запроса/ответа

Файл `index.php` содержит не так уж много кода. Он выполняет три основные функции.

Во-первых, он загружает файл автозагрузки утилиты `Composer`, в котором регистрируются все загружаемые ею зависимости.

COMPOSER И LARAVEL

Основные функциональные возможности `Laravel` разбиты на ряд компонентов, которые расположены в пространстве имен `Illuminate` и загружаются в каждое приложение `Laravel` с помощью утилиты `Composer`. Фреймворк также подгружает довольно много пакетов из `Symfony` и ряда других, разработанных сообществом. Таким образом, `Laravel` можно в равной мере считать и фреймворком, и просто тщательно подобранным набором компонентов.

Во-вторых, он запускает начальную загрузку `Laravel`, создавая контейнер приложения (об этом рассказано в главе 11) и регистрируя несколько основных служб (включая ядро, о котором мы поговорим чуть позже).

Наконец, он создает экземпляр ядра, создает запрос, представляющий веб-запрос текущего пользователя, и передает его для обработки. Ядро выдает в качестве ответа объект `Illuminate Response`, который возвращается файлом `index.php` конечному пользователю. После этого ядро уничтожает запрос страницы.

Ядро `Laravel`

Ядро — это основной маршрутизатор каждого приложения `Laravel`, отвечающий за прием запроса пользователя, обработку запроса с использованием `middleware`, обработку исключений, передачу страничному маршрутизатору и возвращение ответа. На самом деле имеется два ядра, но только одно используется для каждого запроса страницы. Один из этих маршрутизаторов обрабатывает веб-запросы (HTTP-ядро),

а второй — запросы консоли, утилит `сron` и `Artisan` (ядро консоли). У обоих ядер есть метод `handle()`, который обеспечивает получение объекта `Illuminate Request` и возвращение объекта `Illuminate Response`.

Ядро выполняет перед каждым запросом необходимые начальные загрузки, включая проверку того, в какой среде (промежуточной, локальной, эксплуатационной или др.) должен выполняться текущий запрос, и запуск всех сервис-провайдеров. HTTP-ядро дополнительно составляет список `middleware`, выступающего в качестве обертки для каждого запроса, включая основное `middleware`, отвечающее за сессии и CSRF-защиту.

Сервис-провайдеры

Хотя в этих программах начальной загрузки есть определенная доля процедурного кода, почти весь код начальной загрузки Laravel представлен в виде так называемых *сервис-провайдеров*. Сервис-провайдер — это класс, инкапсулирующий логику, которую должны запускать различные части приложения для начальной загрузки своей базовой функциональности.

Например, класс `AuthServiceProvider` производит начальную загрузку всех регистрационных данных, необходимых для системы аутентификации Laravel, а класс `RouteServiceProvider` совершает начальную загрузку системы маршрутизации.

Усвоить смысл концепции сервис-провайдеров будет легче, если думать о ней следующим образом: многие компоненты приложения имеют код начальной загрузки, который должен выполняться на этапе инициализации приложения. Сервис-провайдеры представляют собой инструмент для группировки такого кода начальной загрузки в связанные классы. Если у вас есть код, который нужно выполнять *для подготовки* к запуску кода приложения, то велика вероятность, что этот код следует сделать сервис-провайдером.

Например, если функция, над которой вы работаете, требует регистрации определенных классов в контейнере (см. главу 11), следует создать сервис-провайдер для этой части функциональности, что-то вроде `GitHubServiceProvider` или `MailerServiceProvider`.

Методы `boot()` и `register()` и отложенная регистрация сервис-провайдеров

Сервис-провайдеры имеют два важных метода: `boot()` и `register()`. Можно использовать интерфейс `DeferrableProvider` (в версии 5.8 и следующих) и свойство `$defer` (в версии 5.7 и предыдущих). Применяются они следующим образом.

Сначала вызываются все методы `register()` сервис-провайдеров. Здесь осуществляется связывание классов и псевдонимов с контейнером. В методах `register()` не следует размещать код, использующий уже полностью загруженное приложение.

Затем вызываются все методы `boot()` сервис-провайдеров для выполнения остальной части начальной загрузки. Например, чтобы привязать слушатель событий или определить маршруты — все, что использует полностью загруженное приложение Laravel.

Если ваш сервис-провайдер только регистрирует привязки к контейнеру (то есть обучает контейнер, как следует разрешать определенный класс или интерфейс), не выполняя какой-либо другой начальной загрузки, то вы можете отложить его регистрацию. Значит, она не будет выполняться до явного запроса из контейнера одной из соответствующих привязок. Это может ускорить среднее время начальной загрузки приложения.

Чтобы отложить регистрацию сервис-провайдера, следует сначала реализовать интерфейс `Illuminate\Contracts\Support\DeferrableProvider` (в версии 5.8 и следующих) или добавить для сервис-провайдера защищенное свойство `$defer` со значением `true` (в версии 5.7 и предыдущих). Затем указать для сервис-провайдера метод `provides()` (во всех версиях), возвращающий список обеспечиваемых провайдером привязок (пример 10.1).

Пример 10.1. Отложенная регистрация сервис-провайдера

```
...
use Illuminate\Contracts\Support\DeferrableProvider;

class GitHubServiceProvider extends ServiceProvider implements DeferrableProvider
{
    public function provides()
    {
        return [
            GitHubClient::class,
        ];
    }
}
```



Другие способы использования сервис-провайдеров

Сервис-провайдеры также обладают рядом методов и параметров конфигурации, позволяющих обеспечить расширенные функциональные возможности для конечного пользователя при публикации провайдера в составе пакета `Composer`. Чтобы лучше узнать, как это можно использовать, ознакомьтесь с определением сервис-провайдера в исходном коде Laravel по адресу <http://bit.ly/2HEEC1t>.

Зная, как выполняется начальная загрузка приложения, посмотрим, что собой представляет объект `Request` — самый важный результат начальной загрузки.

Объект Request

Класс `Illuminate\Http\Request` — специфичное для Laravel расширение класса `Symfony\HttpFoundation\Request`.

SYMFONY HTTPFOUNDATION

Набор классов `HttpFoundation` фреймворка `Symfony` используется практически всеми современными PHP-фреймворками. Это самый популярный и мощный набор абстракций для представления в PHP-коде HTTP-запросов, HTTP-ответов, заголовков, cookie-файлов и многого другого.

Объект `Request` служит для представления всей необходимой релевантной информации о HTTP-запросе пользователя.

В нативном PHP-коде можно увидеть переменные `$_SERVER`, `$_GET`, `$_POST` и другие глобальные переменные в сочетании с обрабатывающей логикой для получения информации о запросе текущего пользователя. Какие файлы загрузил на сервер пользователь? Какой у них IP-адрес? Какие поля они опубликовали? Эти данные рассеяны по различным конструкциям языка и по всему вашему коду, что затрудняет их понимание и моделирование.

Вместо этого объект `Request` фреймворка `Symfony` сводит всю необходимую для представления отдельного HTTP-запроса информацию в один объект с удобными методами для легкого извлечения из него нужной информации. Объект `Illuminate Request` предлагает еще больше удобных методов для получения информации о запросе, представляемом этим объектом.



Захват запроса

Вам вряд ли когда-либо потребуется это в приложении `Laravel`, но при необходимости можно захватить объект `Illuminate Request` непосредственно из глобальных переменных языка PHP, используя метод `capture()`:

```
$request = Illuminate\Http\Request::capture();
```

Получение объекта Request в Laravel

`Laravel` создает внутренний объект `Request` для каждого запроса, получить доступ к которому можно несколькими способами.

Первый — задать подсказку этого типа в любом конструкторе или методе, распознаваемом контейнером (см. главу 11). Это означает, что вы можете явно

указать тип `Request` в методе контроллера или сервис-провайдере, как показано в примере 10.2.

Пример 10.2. Указание подсказки типа в распознаваемом контейнером методе для принятия объекта `Request`

```
...
use Illuminate\Http\Request;

class PeopleController extends Controller
{
    public function index(Request $request)
    {
        $allInput = $request->all();
    }
}
```

Второй способ — воспользоваться глобальным хелпером `request()`, вызывая для него методы (например, `request()->input()`) или непосредственно его сам для получения экземпляра `$request`:

```
$request = request();
$allInput = $request->all();
// или
$allInput = request()->all();
```

Третий — получить экземпляр класса `Request`, используя глобальный метод `app()`. При этом можно передать методу полностью определенное имя класса или сокращенный псевдоним `request`:

```
$request = app(Illuminate\Http\Request::class);
$request = app('request');
```

Получение основной информации о запросе

Вы знаете, как получить экземпляр класса `Request`. Теперь поговорим, что с ним можно сделать. Поскольку основным назначением объекта `Request` является представление текущего HTTP-запроса, функциональность класса `Request` призвана обеспечивать легкое получение необходимой информации о текущем запросе.

Я разбил описываемые здесь методы на ряд категорий. Однако эти категории достаточно условны и часто перекрывают друг друга — так, например, параметры запроса можно легко отнести и к категории «пользователь и статус запроса», и к основному пользовательскому вводу. Эти категории призваны лишь помочь вам запомнить доступные методы, после чего их можно будет отбросить.

Здесь перечислены далеко не все методы объекта `Request`, а только наиболее употребительные.

Базовый пользовательский ввод

Методы для получения базового пользовательского ввода позволяют легко получать напрямую вводимые пользователем данные, например, путем подтверждения формы или AJAX-компонента. Под пользовательским вводом здесь подразумеваются входные данные строк запроса (GET), отправляемых форм (POST) или формата JSON. Доступны следующие методы для получения базового пользовательского ввода.

- ❑ `all()`. Возвращает массив из всех введенных пользователем данных.
- ❑ `input(имяПоля)`. Возвращает введенное пользователем значение одного поля ввода.
- ❑ `only(имяПоля | [массив, из, имен, полей])`. Возвращает массив из всех введенных пользователем данных для указанных полей ввода.
- ❑ `except(имяПоля | [массив, из, имен, полей])`. Возвращает массив из всех введенных пользователем данных, за исключением указанных полей ввода.
- ❑ `exists(имяПоля)`. Возвращает логическое значение, указывающее, присутствует ли указанное поле во входных данных. Имеет псевдоним `has()`.
- ❑ `filled(имяПоля)`. Возвращает логическое значение, указывающее, присутствует ли указанное поле во входных данных и является ли оно заполненным (то есть имеет ли оно значение).
- ❑ `json()`. Возвращает объект `ParameterBag`, если странице было отправлено JSON-сообщение.
- ❑ `json(имяКлюча)`. Извлекает значение указанного ключа из JSON-сообщения, отправленного странице.

ОБЪЕКТ PARAMETERBAG

Используя Laravel, вы иногда будете сталкиваться с объектом `ParameterBag`. Этот класс представляет собой нечто вроде ассоциативного массива. Вы можете получить значение определенного ключа с помощью метода `get()`:

```
echo $bag->get('name');
```

Можно воспользоваться методом `has()` для проверки на наличие в массиве определенного ключа, `all()` — для получения массива, содержащего все ключи и значения, `count()` — для подсчета количества элементов и `keys()` — для получения массива, содержащего только ключи.

В примере 10.3 представлено несколько вариантов использования методов для получения из запроса вводимой пользователем информации.

Пример 10.3. Получение из запроса базового пользовательского ввода

```
// Форма
<form method="POST" action="/form">
  @csrf
  <input name="name"> Name<br>
  <input type="submit">
</form>

// Маршрут получения формы
Route::post('form', function (Request $request) {
    echo 'name is ' . $request->input('name') . '<br>';
    echo 'all input is ' . print_r($request->all()) . '<br>';
    echo 'user provided email address: ' . $request->has('email') ? 'true' :
'false';
});
```

Пользователь и статус запроса

Методы, относящиеся к пользователю и статусу запроса, предоставляют входные данные, которые не были напрямую введены пользователем в форме.

- ❑ **method()**. Возвращает метод (GET, POST, PATCH и т. д.), используемый для доступа к данному маршруту.
- ❑ **path()**. Возвращает путь (без домена), применяемый для доступа к данной странице. Например, для страницы <http://www.myapp.com/abc/def> будет возвращен путь `abc/def`.
- ❑ **url()**. Возвращает URL-адрес (с доменом), используемый для доступа к данной странице. Например, для страницы <http://www.myapp.com/abc> будет возвращен URL-адрес `http://www.myapp.com/abc`.
- ❑ **is()**. Возвращает логическое значение, указывающее, имеется ли частичное совпадение текущего запроса страницы с предоставленной строкой. Например, оценка на совпадение `$request->is('*b*')`, где `*` обозначает любые символы, выдаст положительный результат для запроса `/a/b/c`. Данный метод использует собственный синтаксический анализатор регулярных выражений, определенный в методе `Str::is()`.
- ❑ **ip()**. Возвращает IP-адрес пользователя.
- ❑ **header()**. Возвращает массив заголовков (например, `['accept-language' => ['en-US,en;q=0.8']]`) или только указанный заголовок, если в качестве параметра передается имя заголовка.
- ❑ **server()**. Возвращает массив переменных, обычно хранящихся в переменной `$_SERVER` (таких как `REMOTE_ADDR`), или возвращает только ее значение, если передается имя отдельной переменной.
- ❑ **secure()**. Возвращает логическое значение, указывающее, была ли данная страница загружена с использованием протокола HTTPS.

- ❑ `ajax()`. Возвращает логическое значение, указывающее, был ли данный запрос страницы загружен с использованием плагина Ajax.
- ❑ `wantsJson()`. Возвращает логическое значение, указывающее, присутствуют ли типы содержимого `/json` в заголовках **Accept** данного запроса.
- ❑ `isJson()`. Возвращает логическое значение, указывающее, присутствуют ли типы содержимого `/json` в заголовке **Content-Type** данного запроса страницы.
- ❑ `accepts()`. Возвращает логическое значение, указывающее, принимает ли данный запрос страницы содержимое указанного типа.

Файлы

Все рассмотренные входные данные вводятся пользователем напрямую (как в случае методов `all()` и `input()`) либо определяются браузером/ссылающимся сайтом (например, в случае метода `ajax()`). Ввод файлов похож на прямой пользовательский ввод, но обрабатывается по-другому.

- ❑ `file()`. Возвращает массив из всех загруженных на сервер файлов или только один файл, если передается ключ (имя поля для загрузки файла на сервер).
- ❑ `allFiles()`. Возвращает массив из всех загруженных на сервер файлов. В отличие от `file()` имя этого метода лучше отражает его смысл.
- ❑ `hasFile()`. Возвращает логическое значение, указывающее, был ли загружен на сервер соответствующий указанному ключу файл.

Каждый загружаемый на сервер файл представляется экземпляром класса `Symfony\Component\HttpFoundation\File\UploadedFile`, который предлагает ряд инструментов для проверки, обработки и сохранения загружаемых файлов.

Способы обработки загружаемых файлов будут подробно рассмотрены в главе 14.

Долговременное хранение

Объект запроса также может предлагать функции взаимодействия с сессией. Хотя это лишь небольшая часть относящейся к сессиям функциональности, некоторые из этих методов особенно актуальны для текущего запроса страницы.

- ❑ `flash()`. Сохраняет в сессии пользовательский ввод текущего запроса для следующего извлечения. Эти данные сохраняются в сессии, но исчезают после следующего запроса.
- ❑ `flashOnly()`. Сохраняет пользовательский ввод текущего запроса, соответствующий ключам, содержащимся в предоставленном массиве.
- ❑ `flashExcept()`. Сохраняет пользовательский ввод текущего запроса, за исключением данных, соответствующих ключам, содержащимся в предоставленном массиве.

- ❑ `old()`. Возвращает массив, содержащий весь ранее сохраненный пользовательский ввод. Если передается ключ — значение указанного ключа, при условии, что оно было ранее сохранено.
- ❑ `flush()`. Стирает ранее сохраненный пользовательский ввод.
- ❑ `cookie()`. Извлекает из запроса все cookie-файлы или, если передается ключ, извлекает только указанный cookie-файл.
- ❑ `hasCookie()`. Возвращает логическое значение, указывающее, присутствует ли в запросе cookie-файл, соответствующий указанному ключу.

Методы `flash*()` и `old()` предназначены для сохранения пользовательского ввода и его последующего извлечения, часто после выполнения проверки с отбраковкой некорректных данных.

Объект Response

Наряду с объектом **Request** мы можем использовать аналогичный объект **Illuminate Response**, который служит для представления ответа, отправляемого приложением конечному пользователю, вместе со всеми заголовками, cookie-файлами, содержащим и всем, что служит для отправки указаний в отношении отображения страницы браузеру конечного пользователя.

Как и **Request**, класс `Illuminate\Http\Response` является расширением класса `Symfony`, а именно `Symfony\Component\HttpFoundation\Response`. Это базовый класс с набором свойств и методов, обеспечивающих представление и отображение ответа. Класс **Illuminate Response** дополняет этот набор рядом удобных сокращенных псевдонимов.

Использование и создание объектов Response в контроллерах

Прежде чем переходить настраивать объекты **Response**, остановимся на их обычном использовании.

Любой объект **Response**, возвращаемый из определения маршрута, в конечном итоге преобразуется в HTTP-ответ. Он может определять конкретные заголовки или содержимое, cookie-файлы или что-либо еще. Но в любом случае будет преобразован в ответ, который способны проанализировать браузеры пользователей.

Рассмотрим предельно простой ответ, показанный в примере 10.4.

Пример 10.4. Простейший HTTP-ответ

```
Route::get('route', function () {
    return new Illuminate\Http\Response('Hello!');
});
```

```
// То же самое, но с использованием глобальной функции:
Route::get('route', function () {
    return response('Hello!');
});
```

Мы создаем ответ, передаем ему некоторую основную информацию, а затем возвращаем его. Можно настроить HTTP-статус, заголовки, cookie-файлы и многое другое, как показано в примере 10.5.

Пример 10.5. Простой HTTP-ответ с настроенным статусом и заголовками

```
Route::get('route', function () {
    return response('Error!', 400)
        ->header('X-Header-Name', 'header-value')
        ->cookie('cookie-name', 'cookie-value');
});
```

Определение заголовков

Для определения заголовка ответа нужен текущий метод `header()`, показанный в примере 10.5. В качестве первого параметра передается имя заголовка, второго — его значение.

Добавление cookie-файлов

При необходимости можно определить cookie-файлы непосредственно в объекте `Response`. О работе с cookie-файлами в Laravel подробно рассказано в главе 14. Однако в простейшем случае cookie-файлы можно добавить в ответ так, как показано в примере 10.6.

Пример 10.6. Прикрепление к ответу cookie-файла

```
return response($content)
    ->cookie('signup_dismissed', true);
```

Специализированные типы ответов

Можно использовать ряд специальных типов ответов для представлений, скачивания и отображения файлов, а также формата JSON. Каждый из таких ответов представляет собой предопределенный макрос, призванный упростить применение определенных шаблонов для заголовков или структуры содержимого.

Ответы типа view

В главе 4 было показано, как использовать глобальный хелпер `view()` для возвращения шаблона — `view('имя.нужного.представления')` — или чего-либо аналогичного. Однако если при возвращении представления нужно настроить заголовки, HTTP-статус или что-то еще, поможет тип ответа `view()`, как показано в примере 10.7.

Пример 10.7. Использование типа ответа `view()`

```
Route::get('/', function (XmlGetterService $xml) {
    $data = $xml->get();
    return response()
        ->view('xml-structure', $data)
        ->header('Content-Type', 'text/xml');
});
```

Ответы типа `download`

Иногда нужно, чтобы приложение заставляло браузер пользователя выполнить скачивание какого-то файла — это может быть как созданный в Laravel файл, так и извлеченный из базы данных или некоторого другого защищенного хранилища файл. Это достаточно просто реализуется с помощью типа ответа `download()`.

В качестве обязательного первого параметра передается путь к файлу, скачивание которого должен выполнить браузер. Если это генерируемый файл, его нужно где-то временно сохранить.

Как необязательный второй параметр передается имя скачиваемого файла (например, `export.csv`). Если вы не отправите здесь строку, она будет сгенерирована автоматически. В качестве необязательного третьего параметра можно передать массив заголовков. Как можно использовать тип ответа `download()`, показано в примере 10.8.

Пример 10.8. Использование типа ответа `download()`

```
public function export()
{
    return response()
        ->download('file.csv', 'export.csv', ['header' => 'value']);
}

public function otherExport()
{
    return response()->download('file.pdf');
}
```

Если нужно удалять исходный файл с диска после возвращения ответа `download`, пристыкуйте к методу `download()` метод `deleteFileAfterSend()`:

```
public function export()
{
    return response()
        ->download('file.csv', 'export.csv')
        ->deleteFileAfterSend();
}
```

Ответы типа `file`

Ответ типа `file` аналогичен ответу типа `download`, но вместо принудительного скачивания файла позволяет браузеру его отобразить, что часто требуется для изображений и PDF-файлов.

В качестве обязательного второго параметра передается имя файла, а как необязательный второй параметр можно отправить массив заголовков (пример 10.9).

Пример 10.9. Использование типа ответа `file()`

```
public function invoice($id)
{
    return response()->file("./invoices/{$id}.pdf", ['header' => 'value']);
}
```

Ответы JSON

Несмотря на то что программирование ответов в формате JSON не представляет особых сложностей, в силу частого использования для них тоже предусмотрен специальный тип ответа.

Ответы JSON преобразуют передаваемые данные в формат JSON (с помощью метода `json_encode()`) и присваивают заголовку `Content-Type` значение `application/json`. При желании можно создавать ответ не в формате JSON, а в формате JSONP, применяя метод `setCallback()`, как показано в примере 10.10.

Пример 10.10. Использование типа ответа `json()`

```
public function contacts()
{
    return response()->json(Contact::all());
}

public function jsonpContacts(Request $request)
{
    return response()
        ->json(Contact::all())
        ->setCallback($request->input('callback'));
}

public function nonEloquentContacts()
{
    return response()->json(['Tom', 'Jerry']);
}
```

Ответы-перенаправления

Перенаправления отличаются от рассмотренных выше специальных типов ответов тем, что достаточно редко вызываются хелпером `response()`. При этом они тоже являются лишь разновидностью ответа. Перенаправления, возвращаемые из маршрута Laravel, отправляют пользователю перенаправление (часто с кодом 301) на другую страницу или обратно на предыдущую страницу.

Ничто не мешает вызывать перенаправление хелпером `response()`, как в выражении `return response()->redirectTo('/')`. Обычно для этого используются специальные глобальные хелперы.

Так, вы можете создавать ответы-перенаправления с помощью глобальных функций `redirect()` и `back()` (сокращенный псевдоним для вызова `redirect()->back()`).

Как и в случае большинства других глобальных хелперов, вы можете либо передавать параметры `redirect()`, либо использовать хелпер для получения экземпляра соответствующего класса, а затем пристыковывать к нему цепочку вызовов методов. В случае, когда вы просто передаете параметры, не используя цепочку методов, хелпер `redirect()` действует аналогично вызову `redirect()->to()` — принимает строку и выполняет перенаправление по URL-адресу в этой строке (пример 10.11).

Пример 10.11. Примеры использования глобального хелпера `redirect()`

```
return redirect('account/payment');
return redirect()->to('account/payment');
return redirect()->route('account.payment');
return redirect()->action('AccountController@showPayment');

// Если производится перенаправление на внешний домен
return redirect()->away('https://tighten.co');

// Если требуется предоставить параметры для именованного маршрута или контроллера
return redirect()->route('contacts.edit', ['id' => 15]);
return redirect()->action('ContactsController@edit', ['id' => 15]);
```

Можно выполнить перенаправление назад на предыдущую страницу, что особенно полезно при обработке и проверке пользовательского ввода. Типичный шаблон такого перенаправления в контексте проверки ввода показан в примере 10.12.

Пример 10.12. Перенаправление назад с вводом

```
public function store()
{
    // Если проверка дает отрицательный результат...
    return back()->withInput();
}
```

Вы можете совмещать перенаправление с сохранением данных в сессии. Это часто используется при выводе сообщений об ошибках и успешном выполнении действий, как показано в примере 10.13.

Пример 10.13. Перенаправление с сохранением данных

```
Route::post('contacts', function () {
    // Сохранение контакта

    return redirect('dashboard')->with('message', 'Contact created!');
});

Route::get('dashboard', function () {
    // Получение сохраненных данных из сессии — обычно выполняется
    // в шаблоне Blade
    echo session('message');
});
```


Собственные макросы ответов

Можно создавать собственные типы ответов, используя *макросы*, и при этом определить ряд изменений, вносимых в ответ и предоставляемое им содержимое.

Чтобы посмотреть, как это делается, попробуем создать свой вариант типа ответа `json()`. Обычно для таких привязок создается собственный сервис-провайдер, но здесь мы просто поместим ее в `AppServiceProvider`, как показано в примере 10.14.

Пример 10.14. Создание собственного макроса ответа

```
...
class AppServiceProvider
{
    public function boot()
    {
        Response::macro('myJson', function ($content) {
            return response(json_encode($content))
                ->withHeaders(['Content-Type' => 'application/json']);
        });
    }
}
```

После этого данный макрос можно использовать точно так же, как предопределенный макрос `json()`:

```
return response()->myJson(['name' => 'Sangeetha']);
```

Код вернет ответ с телом, содержащим данный массив, закодированный в формате JSON, и заголовком `Content-Type` с подходящим для формата JSON значением.

Интерфейс Responsable

5.5 Если нужно настроить способ отправки ответов, а при этом вам не хватает возможностей макроса в плане объема/организации кода, или если надо, чтобы некоторые из ваших объектов можно было возвращать как «ответ» с их собственной логикой отображения, то можно использовать интерфейс `Responsable` (впервые появился в Laravel 5.5).

Интерфейс `Responsable`, а точнее, `Illuminate\Contracts\Support\Responsable` предписывает, чтобы реализующие его классы имели метод `toResponse()`. Этот метод должен возвращать объект `Illuminate Response`. Как создать объект `Responsable`, показано в примере 10.15.

Пример 10.15. Создание простого объекта Responsable

```
...
use Illuminate\Contracts\Support\Responsable;

class MyJson implements Responsable
{
    public function __construct($content)
```

```
{
    $this->content = $content;
}

public function toResponse()
{
    return response(json_encode($this->content))
        ->withHeaders(['Content-Type' => 'application/json']);
}
```

Затем этот объект можно использовать точно так же, как собственный макрос:

```
return new MyJson(['name' => 'Sangeetha']);
```

Хотя из данного примера можно сделать вывод, что интерфейс `Responsable` требует больших затрат усилий по сравнению с рассмотренными выше макросами ответа, его использование действительно оправдывает себя, когда речь заходит о достаточно сложных манипуляциях с контроллером. Так, например, он часто применяется для создания моделей представлений (или объектов представлений), как показано в примере 10.16.

Пример 10.16. Использование интерфейса `Responsable` для создания объекта представления

```
...
use Illuminate\Contracts\Support\Responsable;

class GroupDonationDashboard implements Responsable
{
    public function __construct($group)
    {
        $this->group = $group;
    }

    public function budgetThisYear()
    {
        // ...
    }

    public function giftsThisYear()
    {
        // ...
    }

    public function toResponse()
    {
        return view('groups.dashboard')
            ->with('annual_budget', $this->budgetThisYear())
            ->with('annual_gifts_received', $this->giftsThisYear());
    }
}
```

В этом контексте применение интерфейса `Responsable` уже действительно имеет смысл — вы переносите сложный код подготовки представления в специально выделенный, *тестируемый* объект, тем самым обеспечивая чистоту кода кон-

троллеров. Вот так будет выглядеть контроллер, использующий данный объект **Responsable**:

```
...
class GroupController
{
    public function index(Group $group)
    {
        return new GroupDonationsDashboard($group);
    }
}
```

Laravel и middleware

Еще раз взглянем на рис. 10.1 в начале этой главы.

Мы знаем, что собой представляют запросы и ответы, но еще не говорили о **middleware**. Возможно, вы уже сталкивались с ним, поскольку это не какая-то уникальная особенность фреймворка **Laravel**, а широко используемый архитектурный шаблон.

Вводная информация о middleware

Концепция **middleware** основана на идее о том, что ваше приложение, подобно луковице или слоеному пирогу, обернуто рядом слоев. Как показано на рис. 10.1, каждый запрос проходит через каждый слой **middleware** на своем пути в приложение, после чего полученный ответ проходит через все слои **middleware** в обратном направлении к конечному пользователю.

Обычно **middleware** не считается составной частью логики приложения и, соответственно, разрабатывается таким образом, чтобы его можно было применять к любому приложению, а не только к тому, над которым вы работаете в данный момент.

Middleware может проверить запрос и, в зависимости от результата этой проверки, либо декорировать, либо отбросить запрос. Это означает, что **middleware** хорошо подходит для таких задач, как ограничение частоты запросов. Так, в данном случае **middleware** может проверять, сколько запросов к заданному ресурсу поступало от определенного IP-адреса за последнюю минуту, и при превышении лимита отправлять обратно код состояния 429 (**Too Many Requests** (Слишком много запросов)).

Поскольку **middleware** также получает доступ к ответу на его пути обратно из приложения, оно хорошо подходит и для декорирования ответов. Так, например, в **Laravel** **middleware** используется для добавления в ответ всей очереди cookie-файлов из заданного цикла запроса/ответа непосредственно перед отправкой ответа конечному пользователю.

Пожалуй, самая важная область применения middleware обусловлена тем фактом, что оно *начинает* и *заканчивает* цикл запроса/ответа. Благодаря этому его удобно использовать для таких вещей, как активирование сессий. Для выполнения РНР-кода нужно, чтобы сессия открывалась как можно раньше и закрывалась как можно позже, и middleware отлично справляется с этой задачей.

Создание собственного middleware

Допустим, что нам требуется middleware, которое бы отклоняло все запросы, использующие HTTP-метод DELETE, а также возвращало cookie-файл для каждого запроса.

Создать собственное middleware можно с помощью специальной команды Artisan. Опробуем ее:

```
php artisan make:middleware BanDeleteMethod
```

Если после этого вы откроете файл `app/Http/Middleware/BanDeleteMethod.php`, то увидите стандартный код, показанный в примере 10.17.

Пример 10.17. Стандартный код middleware

```
...
class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

При изучении концепции middleware труднее всего разобраться с тем, как методу `handle()` удастся представить обработку входящего запроса и исходящего ответа, поэтому пошагово рассмотрим, что делает этот метод.

Принцип действия middleware-метода `handle()`

Middleware представляет собой слои, которые накладываются один поверх другого и в итоге поверх всего приложения. Middleware, которое регистрируется первым, осуществляет *первый* доступ к поступающему запросу, после чего этот запрос последовательно передается каждому следующему middleware и, наконец, приложению. Затем полученный ответ передается по цепочке middleware в обратном направлении. И это первое middleware осуществляет *последний* доступ к ответу на его обратном пути.

Предположим, что мы зарегистрировали класс `BanDeleteMethod` в качестве первого middleware. В таком случае передаваемая ему переменная `$request` будет содержать запрос в его исходном виде, еще не затронутый действием каких-либо других middleware. Но что дальше?

Передача этого запроса замыканию `$next()` означает передачу его остальным middleware. `$next()` просто принимает запрос `$request` и передает его методу `handle()` следующего middleware в стеке. Затем он последовательно передается дальше, пока не пройдет через всю цепочку middleware и достигнет приложения.

Как происходит возвращение ответа? Это действительно сложно понять. Приложение возвращает ответ, который передается обратно по цепочке middleware, поскольку каждое из них возвращает свой ответ. Таким образом, используя один и тот же метод `handle()`, middleware может декорировать запрос `$request` и передать его замыканию `$next()`, а затем выполнить определенные действия над переданным ему результатом до его окончательного возвращения конечному пользователю. Чтобы было проще это понять, рассмотрим псевдокод в примере 10.18.

Пример 10.18. Псевдокод, поясняющий процесс вызова middleware

```
...
class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        // В этой точке переменная $request представляет собой исходный запрос
        // пользователя. Давайте для интереса что-нибудь с ним сделаем.
        if ($request->ip() === '192.168.1.1') {
            return response('BANNED IP ADDRESS!', 403);
        }

        // Теперь мы решили принять запрос.
        // Давайте передадим его следующему middleware в стеке.
        // Мы передаем его замыканию $next(), что обеспечивает возвращение
        // ответа после того, как запрос $request пройдет через стек
        // middleware в приложение и ответ приложения пройдет через
        // стек middleware в обратном направлении.
        $response = $next($request);

        // В этой точке мы можем произвести еще одно взаимодействие
        // с ответом непосредственно перед его возвращением пользователю
        $response->cookie('visited-our-site', true);

        // Наконец, мы можем отправить этот ответ конечному пользователю
        return $response;
    }
}
```

Теперь заставим middleware делать то, для чего оно предназначено (пример 10.19).

Пример 10.19. Пример middleware, отклоняющего запросы, использующие метод DELETE

```
...
class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        // Проверка на использование метода DELETE
```

```
if ($request->method() === 'DELETE') {  
    return response(  
        "Get out of here with that delete method",  
        405  
    );  
}  
  
$response = $next($request);  
  
// Присвоение cookie-файла  
$response->cookie('visited-our-site', true);  
  
// Возвращение ответа  
return $response;  
}  
}
```

Привязка middleware

Однако это еще не все. Мы должны зарегистрировать middleware, что можно сделать глобально или для определенных маршрутов.

При этом глобальное middleware применяется ко всем маршрутам, а маршрутное — только к некоторым.

Привязка глобального middleware

Оба вида привязки middleware определяются в файле `app/Http/Kernel.php`. Чтобы добавить middleware в качестве глобального, добавьте имя его класса в свойство `$middleware`, как показано в примере 10.20.

Пример 10.20. Привязка глобальных middleware

```
// app/Http/Kernel.php  
protected $middleware = [  
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,  
    \App\Http\Middleware\BanDeleteMethod::class,  
];
```

Привязка маршрутного middleware

Middleware, предназначенное для определенных маршрутов, может добавляться в качестве маршрутного или в составе группы middleware. Начнем с первого случая.

Маршрутное middleware добавляется в массив `$routeMiddleware` в файле `app/Http/Kernel.php`. Это делается так же, как в случае добавления middleware в массив `$middleware`. Отличие в том, что каждое middleware снабжается ключом, с помощью которого оно будет применяться к конкретному маршруту (пример 10.21).

Пример 10.21. Привязка маршрутного middleware

```
// app/Http/Kernel.php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
    'ban-delete' => \App\Http\Middleware\BanDeleteMethod::class,
];
```

Теперь можно использовать это middleware в своих определениях маршрутов, как показано в примере 10.22.

Пример 10.22. Применение маршрутного middleware в определениях маршрутов

```
// Не имеет особого смысла для нашего текущего примера...
Route::get('contacts', 'ContactsController@index')->middleware('ban-delete');

// Уже имеет смысл для нашего текущего примера...
Route::prefix('api')->middleware('ban-delete')->group(function () {
    // Все маршруты, относящиеся к некоторому API
});
```

Использование групп middleware

В Laravel 5.2 появилась концепция групп middleware. Это фактически предварительно подготовленные комплекты middleware, которых целесообразно сгруппировать в определенных контекстах.



Группы middleware в версиях 5.2 и 5.3

В первых релизах версии 5.2 файл маршрутов по умолчанию `route.php` имел три четко выраженных раздела: корневой маршрут (`/`), не входящий ни в одну группу middleware; группу `web` и группу `api`. Это немного сбивало с толку новых пользователей и означало, что корневой маршрут не имел доступа к сессии или всему остальному, что можно было сделать в middleware.

В более поздних релизах версии 5.2 ситуация упростилась: теперь каждый маршрут в файле `route.php` находился в группе `web`. В версии 5.3 и следующих вы получаете файл `routes/web.php` для веб-маршрутов и `routes/api.php` для маршрутов API. О том, как можно добавить маршруты в другие группы, будет рассказано далее.

По умолчанию Laravel предлагает две группы middleware: `web` и `api`. В первую входит все middleware, которое может применяться практически к каждому запросу страницы, включая middleware для cookie-файлов, сессий и CSRF-защиты. Группа `api` не содержит такого middleware — в нее входит только middleware для ограничения частоты запросов и привязки модели маршрута. Все определено в файле `app/Http/Kernel.php`.

Группы middleware можно применять к маршрутам с помощью текущего метода `middleware()` в точности так, как применяется к маршрутам маршрутное middleware:

```
Route::get('/', 'HomeController@index')->middleware('web');
```

Можно создавать собственные группы middleware, добавлять маршрутное middleware в имеющиеся группы и удалять его из имеющихся групп. Такое добавление выполняется так же, как обычное добавление маршрутного middleware. Отличие в том, что добавление производится в снабженные ключами группы в массиве `$middlewareGroups`.

Каким образом эти группы middleware сопоставляются с двумя стандартными файлами маршрутов? Файл `routes/web.php` обернутывается группой `web`, а файл `routes/api.php` — группой `api`.

Файлы `routes/*` загружаются в провайдере `RouteServiceProvider`. Если мы взглянем на код метода `map()` этого провайдера (пример 10.23), то увидим, что он вызывает методы `mapWebRoutes()` и `mapApiRoutes()`, каждый из которых загружает свою группу файлов, уже обернутых соответствующей группой middleware.

Пример 10.23. Маршрутный сервис-провайдер по умолчанию в Laravel 5.3 и более новых версиях

```
// App\Providers\RouteServiceProvider
public function map()
{
    $this->mapApiRoutes();
    $this->mapWebRoutes();
}

protected function mapApiRoutes()
{
    Route::prefix('api')
        ->middleware('api')
        ->namespace($this->namespace)
        ->group(base_path('routes/api.php'));
}

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

Как видно в примере 10.23, мы используем маршрутизатор для загрузки группы маршрутов, определенной в пространстве имен по умолчанию (`App\Http\Controllers`) и относящейся к группе middleware `web`, а также второй группы маршрутов, принадлежащей к `api`.

Передача параметров middleware

Порой бывают ситуации, когда требуется передавать параметры маршрутному middleware. Например, у вас может быть middleware для аутентификации, которое по-разному ограничивает доступ для типов пользователя `member` и `owner`:

```
Route::get('company', function () {
    return view('company.admin');
})->middleware('auth:owner');
```

Чтобы это работало, необходимо добавить один/несколько параметров в middleware-метод `handle()` и соответствующим образом модифицировать его логику, как показано в примере 10.24.

Пример 10.24. Определение маршрутного middleware, принимающего параметры

```
public function handle($request, $next, $role)
{
    if (auth()->check() && auth()->user()->hasRole($role)) {
        return $next($request);
    }

    return redirect('login');
}
```

Обратите внимание, что при этом можно добавить больше одного параметра в метод `handle()` и передавать несколько параметров в определение маршрута, разделяя их запятыми:

```
Route::get('company', function () {
    return view('company.admin');
})->middleware('auth:owner,view');
```

ОБЪЕКТЫ ЗАПРОСА ФОРМЫ

В этой главе было показано, как внедрять объект `Illuminate Request`, представляющий собой простейшую и наиболее часто используемую разновидность объекта запроса.

Однако вы можете внедрять и расширенную версию объекта `Request`. О том, как выполняется привязка и внедрение пользовательских классов, будет рассказано в главе 11. Однако можно использовать специальную разновидность запросов, которая обладает собственным набором поведений — так называемым запросом формы.

О создании и использовании запросов формы было рассказано в разделе «Запросы формы» на с. 209.

Доверенные прокси-серверы

Если вы попытаетесь использовать какие-либо инструменты Laravel для генерирования URL-адресов внутри приложения, то увидите, что Laravel распознает, с использованием какого протокола — HTTP или HTTPS — был доставлен текущий запрос, и генерирует любые ссылки, используя соответствующий протокол.

Однако это не всегда работает при наличии перед приложением прокси-сервера (например, балансировщика нагрузки или иного сетевого прокси-сервера). Многие прокси-серверы отправляют приложению такие нестандартные заголовки, как `X_FORWARDED_PORT` и `X_FORWARDED_PROTO`, и ожидают, что приложение будет «доверять» этим заголовкам, интерпретировать их и использовать в процессе интерпретации HTTP-запроса. Чтобы Laravel мог правильно истолковывать прокси-вызовы по протоколу HTTPS как защищенные вызовы, а также обрабатывать те дополнительные виды заголовков, которые используются в прокси-запросах, необходимо определить для него соответствующие инструкции.

Вам не нужно, чтобы приложение принимало трафик буквально от *всех* прокси-серверов. Вместо этого оно должно ограничиться рядом доверенных прокси-серверов, и даже от этих прокси-серверов, вероятно, нужно принимать только ряд доверенных заголовков пересылки.

5.6 Начиная с версии Laravel 5.6, пакет TrustedProxy (<http://bit.ly/2HEI3tR>) по умолчанию включается в состав каждой установки Laravel. Но вы можете добавить его в свой комплект инструментов и в том случае, если используете более старую версию. Пакет TrustedProxy позволяет внести определенные источники трафика в белый список, указав их в качестве «доверенных» источников, а также отметить, какие «доверенные» заголовки пересылки следует принимать от этих источников и как их нужно отображать на обычные заголовки.

Чтобы указать, каким прокси-серверам должно доверять ваше приложение, отредактируйте код middleware `App\Http\Middleware\TrustProxies`, добавив IP-адрес балансировщика нагрузки или прокси-сервера в массив `$proxies`, как показано в примере 10.25.

Пример 10.25. Настройка middleware TrustProxies

```
/**
 * Доверенные прокси-серверы данного приложения
 *
 * @var array
 */
protected $proxies = [
    '192.168.1.1',
    '192.168.1.2',
];
```

```
/**
 * Заголовки, используемые для распознавания прокси-серверов
 *
 * @var string
 */
protected $headers = Request::HEADER_X_FORWARDED_ALL;
```

Как видите, массив `$headers` по умолчанию предписывает принимать все заголовки пересылки от доверенных прокси-серверов. Если нужно настроить этот список, ознакомьтесь с документацией фреймворка Symfony доверенных прокси-серверов (<http://bit.ly/2UY7Pri>).

Тестирование

Помимо использования запросов, ответов и middleware в рамках тестирования, выполняемого вами как разработчиком приложения, они активно применяются и самим Laravel.

При проверке приложения вызовами вида `$this->get('/')` вы даете указание среде тестирования приложений Laravel сгенерировать объекты запросов, представляющие указанные вами взаимодействия. Затем они передаются приложению, как если бы это были реальные посещения пользователей. Именно это позволяет точно оценивать работу приложения: оно не подозревает, что за осуществляемым взаимодействием не стоит реальный пользователь.

В этом контексте многие из создаваемых вами утверждений — например, `assertResponseOk()` — являются утверждениями в отношении объекта ответа, сгенерированного средой тестирования приложений. Метод `assertResponseOk()` просто получает объект ответа и проверяет, возвращает ли значение `true` его метод `isOk()`, который, в свою очередь, смотрит, равен ли 200 код состояния этого объекта. При тестировании приложения *все* ведет себя так, как если бы в приложение поступал реальный запрос страницы.

Если нужно напрямую использовать запрос в своих тестах, можно извлечь его из контейнера с помощью инструкции `$request = request()`. Вы также можете создать собственный запрос — конструктор класса `Request` принимает указанные ниже необязательные параметры:

```
$request = new Illuminate\Http\Request(
    $query,      // Массив GET-параметров
    $request,    // Массив POST-параметров
    $attributes, // Массив «атрибутов»; может быть пустым
    $cookies,    // Массив cookie-файлов
    $files,      // Массив файлов
    $server,     // Массив серверов
    $content     // Необработанные данные тела запроса
);
```

Если требуется более серьезный пример, ознакомьтесь с методом фреймворка Symfony для создания нового объекта **Request** на основе глобальных переменных языка PHP `Symfony\Component\HttpFoundation\Request::createFromGlobals()`.

Еще проще вручную создавать объекты **Response**. При этом можно использовать следующие необязательные параметры:

```
$response = new Illuminate\Http\Response(  
    $content, // Содержимое ответа  
    $status,  // HTTP-статус; по умолчанию 200  
    $headers  // Массив заголовков  
);
```

Если при проверке приложения требуется отключить middleware, импортируйте в этот тест трейт `WithoutMiddleware`. Можно отключить middleware для отдельного метода тестирования, используя метод `$this->withoutMiddleware()`.

Резюме

Каждый запрос в приложении Laravel преобразуется в объект **Illuminate Request**, который затем проходит через всю цепочку middleware и обрабатывается. Приложение генерирует объект **Response**, который после передается через всю цепочку middleware обратно и возвращается конечному пользователю.

Объекты **Request** и **Response** отвечают за инкапсуляцию и представление всей релевантной информации о входящем запросе пользователя и исходящем ответе сервера.

Сервис-провайдеры собирают в одном месте связанное поведение привязки и регистрации классов для использования сго приложением.

Middleware обертывает приложение и может отклонить/декорировать любой запрос или ответ.

Сервисный контейнер, или, как его еще называют, контейнер внедрения зависимостей (DI — Dependency Injection), выступает в качестве фундамента почти для всех других возможностей Laravel. Он представляет собой простейший инструмент для привязки и разрешения конкретных экземпляров классов и интерфейсов. В то же время это мощный и всесторонний менеджер сети взаимосвязанных зависимостей. В этой главе мы подробно обсудим, что он собой представляет, как работает и как его можно использовать.



Различные названия контейнеров

В этой книге, документации и других учебных материалах используются различные названия контейнеров, включая следующие.

- Контейнер приложения.
- IoC-контейнер (Inversion of Control — «инверсия управления»).
- Сервисный контейнер.
- DI-контейнер (Dependency Injection — «внедрение зависимостей»).

Хотя все эти названия вполне допустимы и по-своему полезны, следует иметь в виду, что все они подразумевают одно и то же — сервисный контейнер.

Вводная информация о внедрении зависимостей

Внедрение зависимостей означает, что вместо инстанцирования (обновления) зависимости каждого класса внутри класса вы *внедряете* их извне. Чаще используется *внедрение через конструктор*, при котором зависимости объекта внедряются при его создании. Однако возможно и *внедрение через установщик*, при котором класс специально экспонирует метод для внедрения определенной зависимости, и *внедрение через метод*, при котором один или несколько методов ожидают, что их зависимости будут внедряться при их вызове.

В примере 11.1 показан простейший пример наиболее распространенной разновидности DI — внедрения через конструктор.

Пример 11.1. Простейший пример внедрения зависимостей

<?php

```

class UserMailer
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function welcome($user)
    {
        return $this->mailer->mail($user->email, 'Welcome!');
    }
}

```

Данный класс **UserMailer** ожидает, что при его инстанцировании будет внедряться объект типа **Mailer**, после чего его методы будут ссылаться на этот экземпляр.

Основное преимущество внедрения зависимостей в том, что мы можем свободно изменять то, что мы внедряем, имитировать зависимости для целей тестирования и только один раз инстанцировать совместно используемые зависимости.

ИНВЕРСИЯ УПРАВЛЕНИЯ (INVERSION OF CONTROL, IOC)

В контексте внедрения зависимостей иногда упоминают инверсию управления и то, что контейнер **Laravel** называют **IoC**-контейнером.

На самом деле концепция «инверсия управления» очень близка по своему смыслу концепции «внедрение зависимостей». В ее основе лежит идея, что в традиционном программировании код самого нижнего уровня — конкретные классы, экземпляры и процедурный код — управляет тем, какой экземпляр определенного шаблона или интерфейса следует использовать. Например, если вы инстанцируете почтовый сервис во всех применяющих его классах, каждый класс может по своему усмотрению выбрать сервис **Mailgun**, **Mandrill** или **Sendgrid**.

Инверсия управления подразумевает, что мы разворачиваем это управление на 180 градусов, перенося его в противоположный конец приложения. Теперь определение того, какой почтовый сервис следует использовать, будет на самом верхнем и абстрактном уровне приложения, часто в файлах конфигурации. При этом каждый экземпляр и фрагмент низкоуровневого кода будет обращаться к файлам конфигурации, чтобы выяснить, какой почтовый сервис следует применить. Они будут знать только то, что они должны получить некоторый почтовый сервис, не зная точно, какой именно.

Инверсия управления очень уместна при использовании внедрения зависимостей и особенно **DI**-контейнеров, поскольку позволяет вам лишь один раз определять, какой экземпляр интерфейса **Mailer** должен предоставляться при внедрении почтового сервиса в любой применяющий его класс.

Внедрение зависимостей и Laravel

Как можно видеть из примера 11.1, наиболее распространенный шаблон внедрения зависимостей — через конструктор, то есть внедрение зависимостей объекта при его инстанцировании (конструировании).

Возьмем наш класс `UserMailer` из примера 11.1 и попробуем создать и использовать его экземпляр. Как это можно сделать, показано в примере 11.2.

Пример 11.2. Простейший пример внедрения зависимостей вручную

```
$mailer = new MailgunMailer($mailgunKey, $mailgunSecret, $mailgunOptions);
$userMailer = new UserMailer($mailer);

$userMailer->welcome($user);
```

Представим, что нам нужно, чтобы класс `UserMailer` мог регистрировать сообщения и отправлять уведомление в Slack-канал при каждом высылании почтового сообщения. Это показано в примере 11.3. Если при каждом создании нового экземпляра мы будем сами делать всю эту работу, то придется иметь дело с довольно громоздким кодом — особенно если учесть, что при этом еще нужно будет откуда-то получать все эти параметры.

Пример 11.3. Более сложный пример внедрения зависимостей вручную

```
$mailer = new MailgunMailer($mailgunKey, $mailgunSecret, $mailgunOptions);
$logger = new Logger($logPath, $minimumLogLevel);
$slack = new Slack($slackKey, $slackSecret, $channelName, $channelIcon);
$userMailer = new UserMailer($mailer, $logger, $slack);

$userMailer->welcome($user);
```

Представьте, сколько работы потребуется, если писать этот код при каждом создании объекта `UserMailer`. При всех своих достоинствах внедрение зависимостей легко может обеспечить настоящую головную боль.

Глобальный хелпер app()

Кратко рассмотрим самый простой способ извлечения объекта из контейнера: использование хелпера `app()`.

Если вы передадите этой функции строку, представляющую собой полностью определенное имя класса (fully qualified class name, FQCN), например `App\ThingDoer`, или принятый в Laravel сокращенный псевдоним (мы обсудим псевдонимы чуть ниже), то он возвратит экземпляр этого класса:

```
$logger = app(Logger::class);
```

Это самый простой способ взаимодействия с контейнером. При этом создается экземпляр данного класса и аккуратное возвращение его вам, почти как в случае использования вызова `new Logger`, но намного удобнее.

РАЗЛИЧНЫЕ СИНТАКСИСЫ ДЛЯ СОЗДАНИЯ КОНКРЕТНОГО ЭКЗЕМПЛЯРА

Самый простой способ создания конкретного экземпляра любого класса или интерфейса — воспользоваться глобальным хелпером, напрямую передав ему имя класса или интерфейса: `app('FQCN')`.

Однако если вы получили экземпляр контейнера — например, он был внедрен в каком-то месте, или вы находитесь внутри сервис-провайдера и выполнили вызов `$this->app`, или (менее известный прием) получили его, выполнив инструкцию `$container = app()`, — есть несколько способов создать экземпляр после этого.

Самый распространенный способ — вызвать метод `make()`. Вполне подойдет вызов вида `$app->make('FQCN')`. Однако в коде других разработчиков и в документации можно увидеть следующий синтаксис: `$app['FQCN']`. Не беспокойтесь. Этот синтаксис — просто другой способ написания.

Хотя данный способ создания экземпляра класса `Logger` кажется достаточно простым, как вы, вероятно, заметили, в примере 11.3 классу `$logger` передаются два параметра: `$logPath` и `$minimumLogLevel`. Как контейнер узнает, что следует передавать в данном случае?

Никак. Вы можете использовать глобальный хелпер `app()` для создания экземпляра, не имеющего параметров класса в его конструкторе, но в таком месте можно вызвать `new Logger`. Применение контейнера действительно оправдывает себя, когда код конструктора достаточно сложный. Пора разобраться с тем, каким образом контейнер может выяснить, как следует создавать экземпляр класса в случае конструктора с параметрами.

Как осуществляется привязка к контейнеру

Прежде чем мы углубимся в дальнейшее обсуждение класса `Logger`, взгляните на пример 11.4.

Пример 11.4. Автоматическое внедрение в Laravel

```
class Bar
{
    public function __construct() {}
}
```



```
class Baz
{
    public function __construct() {}
}

class Foo
{
    public function __construct(Bar $bar, Baz $baz) {}
}

$foo = app(Foo::class);
```

Этот код похож на код из примера 11.11 с почтовым сервисом. Отличие в том, что здесь обе зависимости (**Bar** и **Baz**) настолько просты, что контейнер может разрешить их без дополнительной информации. Контейнер читает подсказки типов в конструкторе класса **Foo**, разрешает экземпляры классов **Bar** и **Baz**, а затем внедряет их в новый экземпляр класса **Foo** при его создании. Это называется *автоматическим внедрением*: экземпляры классов разрешаются на основе подсказок типов без необходимости для разработчика выполнять явную привязку этих классов в контейнере.

Автоматическое внедрение означает, что если класс не был явно привязан к контейнеру (как классы **Foo**, **Bar** или **Baz** в данном случае), но контейнер и без этого может выяснить, как его следует разрешить, то контейнер разрешит этот класс. Это значит, что из контейнера могут разрешаться и любые классы без зависимостей в конструкторе (**Bar/Baz**) и с зависимостями, которые способен разрешить контейнер (такой как **Foo**).

Как следствие, нам требуется выполнить привязку только тех классов, у которых в конструкторе неразрешимые параметры — в частности, из примера 11.3 класс **\$logger**, который имеет параметры, представляющие путь к логу и уровень ведения лога.

Для работы с такими классами следует разобраться с тем, как можно явно привязать что-либо к контейнеру.

Привязка классов к контейнеру

Привязка класса к контейнеру **Laravel**, по сути, дает контейнеру следующее указание: «Когда разработчик будет запрашивать экземпляр класса **Logger**, нужно выполнять вот этот код для создания экземпляра с необходимыми параметрами и зависимостями и его последующего корректного возвращения».

Мы указываем контейнеру, что при запрашивании этой конкретной строки (обычно представляющей собой полное имя класса (FQCN)) он должен разрешить ее вот таким образом.

Привязка к замыканию

Посмотрим, как осуществляется привязка к контейнеру. Обратите внимание, что привязку к контейнеру следует выполнять в методе `register()` сервис-провайдера (пример 11.5).

Пример 11.5. Простейший пример привязки к контейнеру

```
// В любом сервис-провайдере (например, в провайдере LoggerServiceProvider)
public function register()
{
    $this->app->bind(Logger::class, function ($app) {
        return new Logger('\log\path\here', 'error');
    });
}
```

В этом примере следует отметить несколько важных моментов. Мы делаем вызов `$this->app->bind()`. `$this->app` — это экземпляр контейнера, который всегда доступен в каждом сервис-провайдере. Метод `bind()` контейнера служит для выполнения привязки к контейнеру.

В качестве первого параметра методу `bind()` передается «ключ», к которому выполняется привязка. В данном случае мы использовали FQCN-имя класса. Второй параметр может быть разным. Зависит от того, что именно нужно делать, но это должно быть *что-то* показывающее контейнеру, как следует разрешать экземпляр привязываемого ключа.

Так, в данном примере в качестве второго параметра передается замыкание. Поэтому теперь при каждом выполнении вызова `app(Logger::class)` будет выдаваться результат этого замыкания. Замыканию передается экземпляр самого контейнера (`$app`). Если разрешаемый вами класс имеет зависимости, которые требуется разрешать из контейнера, можете использовать их в своем определении, как показано в примере 11.6.

Пример 11.6. Использование переданного экземпляра `$app` в коде привязки к контейнеру

```
// Обратите внимание, что данная привязка не несет никакой практической
// пользы, поскольку все, что она делает, уже обеспечено автоматическим
// внедрением в контейнер
$this->app->bind(UserMailer::class, function ($app) {
    return new UserMailer(
        $app->make(Mailer::class),
        $app->make(Logger::class),
        $app->make(Slack::class)
    );
});
```

Каждый раз, когда вы будете запрашивать новый экземпляр класса, это замыкание будет выполняться повторно с возвращением нового результата.

Привязка одиночек, псевдонимов и экземпляров

Если вам нужно, чтобы результат замыкания привязки кэшировался, чтобы замыкание не выполнялось повторно при каждом запрашивании экземпляра, то это соответствует шаблону «Одиночка» (Singleton), который можно реализовать с помощью вызова `$this->app->singleton()`. Как это выглядит, показано в примере 11.7.

Пример 11.7. Привязка одиночки к контейнеру

```
public function register()
{
    $this->app->singleton(Logger::class, function () {
        return new Logger('\log\path\here', 'error');
    });
}
```

Если у вас уже есть экземпляр (Instance) возвращаемого одиночкой объекта, то вы можете получить сходное поведение, как показано в примере 11.8.

Пример 11.8. Привязка к контейнеру существующего экземпляра класса

```
public function register()
{
    $logger = new Logger('\log\path\here', 'error');
    $this->app->instance(Logger::class, $logger);
}
```

Если нужно использовать один класс как псевдоним (Alias) другого класса, привязать класс к сокращенному псевдониму или, наоборот, привязать сокращенный псевдоним к классу, то можно передать две строки (пример 11.9).

Пример 11.9. Назначение псевдонимов для классов и строк

```
// Когда запрашивается "Logger", выдается FirstLogger
$this->app->bind(Logger::class, FirstLogger::class);

// Когда запрашивается "log", выдается FirstLogger
$this->app->bind('log', FirstLogger::class);

// Когда запрашивается "log", выдается FirstLogger
$this->app->alias(FirstLogger::class, 'log');
```

Обратите внимание, что такие сокращенные псевдонимы широко используются в ядре Laravel. Это обеспечивает для классов, предоставляющих базовые функциональные возможности, систему псевдонимов из таких легко запоминающихся ключей, как `log`.

Привязка конкретного экземпляра к интерфейсу

В точности так же, как выполняется привязка класса к другому классу или сокращенному псевдониму, можно привязать к интерфейсу. Это чрезвычайно полезная возможность, поскольку теперь вместо имен классов можно указывать интерфейсы в подсказках типов, как показано в примере 11.10.

Пример 11.10. Указание интерфейса в подсказке типа и привязка к интерфейсу

```
...
use Interfaces\Mailer as MailerInterface;
class UserMailer
{
    protected $mailer;

    public function __construct(MailerInterface $mailer)
    {
        $this->mailer = $mailer;
    }
}

// Сервис-провайдер
public function register()
{
    $this->app->bind(\Interfaces\Mailer::class, function () {
        return new MailgunMailer(...);
    });
}
```

Теперь вы можете указать интерфейсы **Mailer** и **Logger** в подсказках типов в пределах всего кода, а затем однократно указать в сервис-провайдере, какой именно почтовый сервис или регистратор событий следует использовать во всех этих местах. Это инверсия управления.

Одно из ключевых преимуществ применения этого шаблона — если впоследствии вы захотите использовать другой почтовый провайдер вместо Mailgun, то при условии наличия класса почтового сервиса для этого нового провайдера, реализующего интерфейс **Mailer**, достаточно внести лишь одно изменение в своем сервис-провайдере. И оно без каких-либо проблем соответственно поменяет работу всего остального кода.

Контекстная привязка

Иногда требуется изменять способ разрешения интерфейса в зависимости от контекста. Допустим, что события, поступающие из одного места, нужно регистрировать в локальном системном журнале, а поступающие из других мест — с использованием внешнего сервиса. В таком случае следует дать контейнеру указание проводить соответствующее различие (пример 11.11).

Пример 11.11. Контекстная привязка

```
// В сервис-провайдере
public function register()
{
    $this->app->when(FileWrangler::class)
        ->needs(Interfaces\Logger::class)
        ->give(Loggers\Syslog::class);

    $this->app->when(Jobs\SendWelcomeEmail::class)
        ->needs(Interfaces\Logger::class)
        ->give(Loggers\PaperTrail::class);
}
```

Внедрение в конструктор в файлах Laravel

Мы еще не рассмотрели, каким образом контейнер обеспечивает разрешение многих основных рабочих классов приложения. Например, каждый контроллер инстанцируется контейнером. Это означает, что если вам нужно использовать в контроллере экземпляр регистратора событий, то можно указать класс регистратора в подсказках типов конструктора контроллера. Тогда при создании контроллера Laravel разрешит этот класс из контейнера, и этот экземпляр регистратора будет доступен в контроллере. Как это можно сделать, показано в примере 11.12.

Пример 11.12. Внедрение зависимостей в контроллер

```
...
class MyController extends Controller
{
    protected $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }

    public function index()
    {
        // Выполнение некоторых действий
        $this->logger->error('Something happened');
    }
}
```

Контейнер обеспечивает разрешение контроллеров, middleware, заданий очереди, слушателей событий и любых других классов, которые автоматически генерируются фреймворком Laravel на протяжении жизненного цикла приложения. Поэтому при работе с любым из этих классов можно указать зависимости в подсказках типов конструктора и рассчитывать, что они будут внедряться автоматически.

Внедрение через метод

В некоторых местах приложения Laravel читает еще и сигнатуру *методов*, также внедряя в них зависимости.

Наиболее часто внедрение через метод используется в методах контроллеров. Если у вас есть зависимость, которую нужно использовать только в одном методе контроллера, то можно внедрить ее только в этот метод, как показано в примере 11.13.

Пример 11.13. Внедрение зависимостей в метод контроллера

```
...
class MyController extends Controller
{
    // Зависимости метода могут указываться перед или после параметров маршрута
    public function show(Logger $logger, $id)
    {
        // Выполнение некоторых действий
        $logger->error('Something happened');
    }
}
```



Передача неразрешимых параметров конструктора с помощью метода makeWith()

Все основные инструменты для разрешения конкретного экземпляра класса — `app()`, `$container->make()` и т. д. — подразумевают, что все зависимости класса могут быть разрешены без передачи параметров. Но что, если ваш класс принимает в своем конструкторе не зависимость, которую может автоматически разрешить контейнер, а значение? Воспользуйтесь методом `makeWith()`:

```
class Foo
{
    public function __construct($bar)
    {
        // ...
    }
}

$foo = $this->app->makeWith(
    Foo::class,
    ['bar' => 'value']
);
```

Однако это исключение из правил. В большинстве случаев разрешаемые из контейнера классы используют только зависимости, внедряемые через их конструктор.

Вы можете делать то же самое в методе `boot()` сервис-провайдеров, а также произвольным образом вызывать метод любого класса, использующего контейнер, — при этом внедрение через метод будет происходить в этом классе (пример 11.14).

Пример 11.14. Вызов вручную метода класса с помощью метода `call()` контейнера

```
class Foo
{
    public function bar($parameter1) {}
}
// Вызывает метод 'bar' класса 'Foo' с первым параметром 'value'
app()->call('Foo@bar', ['parameter1' => 'value']);
```

Фасады и контейнер

В предыдущих главах мы довольно подробно говорили о фасадах, но не о принципе их работы.

Фасады Laravel представляют собой классы, обеспечивающие простой доступ к основным элементам его функциональности. У них есть две отличительные черты: они доступны в глобальном пространстве имен (так, `\Log` является псевдонимом для `\Illuminate\Support\Facades\Log`) и используют статические методы для доступа к нестатическим ресурсам.

Рассмотрим фасад `Log`, раз уж мы начали рассматривать регистрацию событий в этой главе. В своем контроллере или представлениях можно использовать следующий вызов:

```
Log::alert('Something has gone wrong!');
```

Без использования фасада этот вызов будет выглядеть так:

```
$logger = app('log');
$logger->alert('Something has gone wrong!');
```

Фасады транслируют статические вызовы (то есть любые вызовы методов, выполняемые не в экземпляре, а непосредственно в классе с использованием двойного двоеточия `::`) в обычные вызовы методов в экземплярах.



Импорт пространств имен фасадов

Если вы находитесь в классе, расположенном в некотором пространстве имен, проследите за тем, чтобы перед ним был импортирован фасад:

```
...
use Illuminate\Support\Facades\Log;

class Controller extends Controller
{
    public function index()
    {
        // ...
        Log::error('Something went wrong!');
    }
}
```

Как работают фасады

Возьмем в качестве примера фасад `Cache` и посмотрим, как он в действительности работает.

Для начала откройте класс `Illuminate\Support\Facades\Cache`. Вы увидите примерно такой код, как в примере 11.15.

Пример 11.15. Класс фасада `Cache`

```
<?php

namespace Illuminate\Support\Facades;

class Cache extends Facade
{
    protected static function getFacadeAccessor()
    {
        return 'cache';
    }
}
```

Каждый фасад имеет единственный метод: `getFacadeAccessor()`. Он определяет ключ, используемый `Laravel` для извлечения базового экземпляра фасада из контейнера.

В данном случае каждый вызов фасада `Cache` проксируется на вызов экземпляра псевдонима `cache` из контейнера. Конечно, это не имя реального класса или интерфейса, а один из уже упоминавшихся ранее сокращенных псевдонимов.

Таким образом, происходит следующее:

```
Cache::get('key');

// То же самое, что и...

app('cache')->get('key');
```

Узнать, на какой именно класс указывает аксессор каждого фасада, можно несколькими способами. Однако самый простой — ознакомиться с документацией. В документации по фасадам (<http://bit.ly/2WpJdlu>) вы найдете таблицу, в которой для каждого фасада указано, к какой привязке контейнера (сокращенному псевдониму вроде `cache`) он относится и какой класс возвращает эта привязка. Это выглядит следующим образом.

Фасад	Класс	Привязка сервисного контейнера
App	Illuminate\Foundation\Application	app
...	...	...
Cache	Illuminate\Cache\CacheManager	cache
...	...	...

Наличие этой справочной информации позволяет сделать три вещи.

Во-первых, вы можете выяснить доступные в фасаде методы. Для этого найдите базовый класс фасада и взгляните на его определение. Все публичные методы этого класса можно вызывать в фасаде.

Во-вторых, можно выяснить, как внедрять базовый класс фасада с помощью внедрения зависимостей. Если требуется функциональность фасада, но при этом вы предпочитаете использовать внедрение зависимостей, то укажите базовый класс фасада в подсказках типов либо получите его экземпляр хелпером `app()`. Вызовите те же методы, которые вы бы вызвали в фасаде.

В-третьих, эта информация показывает, как создавать собственные фасады. Сделайте класс фасада, расширяющий класс `Illuminate\Support\Facades\Facade`, и снабдите его методом `getFacadeAccessor()`, возвращающим строку. Эта строка должна представлять собой что-то позволяющее разрешить ваш базовый класс из контейнера. Например, это может быть полное имя класса. После этого нужно зарегистрировать фасад, добавив его в массив `aliases` в файле `config/app.php`, и дело сделано! Это все, что нужно для создания собственного фасада.

Фасады реального времени

5.4 В Laravel 5.4 появилась новая концепция — так называемые *фасады реального времени*. Вместо того чтобы создавать новый класс, чтобы сделать экземплярные методы класса доступными в качестве статических методов, можно добавить префикс `Facades\` к полному имени класса и использовать его *словно фасад*. Как это работает, показано в примере 11.16.

Пример 11.16. Использование фасадов реального времени

```
namespace App;

class Charts
{
    public function burndown()
    {
        // ...
    }
}

<h2>Burndown Chart</h2>
{{ Facades\App\Charts::burndown() }}
```

Нестатический метод `burndown()` становится доступным в качестве статического метода в фасаде реального времени, который создается путем добавления префикса `Facades\` к полному имени класса.

Сервис-провайдеры

Основы работы с сервис-провайдерами были рассмотрены в предыдущей главе (см. подраздел «Сервис-провайдеры» на с. 269). Что самое важное применительно к контейнеру, так это не забывать регистрировать привязки в методе `register()` какого-либо сервис-провайдера.

При этом можно закинуть все незарегистрированные привязки в универсальный провайдер `App\Providers\AppServiceProvider`. Однако в большинстве случаев лучше создать отдельные сервис-провайдеры для всех разрабатываемых групп функциональности и привязать соответствующие классы в методе `register()` каждого отдельного сервис-провайдера.

Тестирование

Возможность использования инверсии управления и внедрения зависимостей обеспечивает в Laravel чрезвычайно широкие возможности тестирования. Например, можно привязывать различный регистратор событий в зависимости от того, введено ли приложение в эксплуатацию или находится на стадии тестирования. Или для облегчения проверки можно перейти с сервиса транзакционных рассылок Mailgun к локальному регистратору электронной почты. Эти два варианта замены применяются так часто, что их даже проще создавать конфигурационными файлами `.env` Laravel. Однако вы можете аналогичным образом заменять любые нужные вам интерфейсы и классы.

Самый простой способ — непосредственно в тесте явно переопределить привязку классов и интерфейсов в нужном месте (пример 11.17).

Пример 11.17. Переопределение привязки в тестах

```
public function test_it_does_something()
{
    app()->bind(Interfaces\Logger, function () {
        return new DevNullLogger;
    });

    // Выполнение определенных действий
}
```

Если нужно переопределить привязку конкретных классов/интерфейсов глобально для своих тестов (что требуется довольно редко), то это можно сделать в методе `setUp()` класса теста или в методе `setUp()` базового теста `TestCase` Laravel, как показано в примере 11.18.

Пример 11.18. Переопределение привязки для всех тестов

```
class TestCase extends \Illuminate\Foundation\Testing\TestCase
{
    public function setUp()
```

```
{
    parent::setUp();

    app()->bind('whatever', 'whatever else');
}
```

При использовании Mockery обычно создается имитация, шпион или заглушка класса, затем этот объект привязывается к контейнеру вместо исходного класса.

Резюме

У сервисного контейнера Laravel много имен, но, как бы вы его ни называли, важно лишь то, что он призван упростить определение способа разрешения строковых имен в виде конкретных экземпляров. В качестве этих строковых имен могут выступать полностью определенные имена классов и интерфейсов или такие сокращенные псевдонимы, как `log`.

Каждая привязка дает приложению указание о том, как при получении некоторого строкового ключа (например, `app('log')`) следует разрешать конкретный экземпляр.

Контейнер достаточно умен для рекурсивного разрешения зависимостей. Поэтому, если вы попытаетесь разрешить экземпляр класса с зависимостями в конструкторе, контейнер попытается разрешить эти зависимости, исходя из их подсказок типов, после чего передаст их в ваш класс и возвратит экземпляр.

Есть несколько способов выполнения привязки к контейнеру, но все эти способы позволяют определить, что следует возвращать при получении некоторой конкретной строки.

Фасады — псевдонимы, позволяющие использовать простые статические вызовы класса, расположенного в корневом пространстве имен, для вызова нестатических методов классов, разрешаемых из контейнера. Фасады реального времени позволяют применять любой класс в качестве фасада, добавляя префикс `Facades\` к полностью определенному имени класса.

12 Тестирование

Большинство разработчиков знает, что тестирование кода — вещь полезная и нужная. Обычно у них также есть некоторое представление, в чем состоит польза этого процесса. Возможно, они прочитали определенные руководства по принципам его выполнения.

Однако знать, *почему* следует выполнять тестирование, и понимать, *как* это следует делать, — далеко не одно и то же. К счастью, такие инструменты, как PHPUnit, Mockery и PHPSpec, предоставляют невероятно широкие возможности настройки процесса тестирования PHP-кода, но, несмотря на это, порой очень сложно сконфигурировать все как надо.

В Laravel по умолчанию встроена интеграция с инструментами PHPUnit (для модульного тестирования), Mockery (для имитирования) и Faker (для создания «поддельных» данных с целью заполнения данными и тестирования). Фреймворк также предлагает собственный простой и мощный набор инструментов для тестирования приложений, позволяющий выполнять «обход» URI-идентификаторов вашего сайта, отправлять формы, проверять коды состояния HTTP, а также выполнять валидацию и проверку утверждений в отношении данных в формате JSON. Помимо этого, Laravel предоставляет надежный инструмент для тестирования клиентской части приложения под названием Dusk, который в числе прочего даже может взаимодействовать и сверяться с вашими JavaScript-приложениями. В данной главе нам предстоит освоить достаточно много материала.

Чтобы упростить для вас первые шаги, вместе со средой тестирования Laravel предоставляется пример теста приложения, который может быть успешно выполнен сразу же после создания нового приложения. То есть не нужно тратить время на настройку среды тестирования, что снимает еще одну преграду на пути к написанию своих тестов.

Основы тестирования

ТЕРМИНОЛОГИЯ ТЕСТИРОВАНИЯ

Если мы возьмем любую достаточно большую группу программистов, то они вряд ли придут к согласию в отношении того, как следует называть различные виды тестов.

В этой книге я буду использовать четыре основных термина.

- *Модульные тесты.* Модульные тесты предназначены для тестирования небольших сравнительно изолированных модулей — обычно классов или методов.
- *Функциональные тесты.* Функциональные тесты служат для проверки того, как отдельные модули взаимодействуют друг с другом и передают сообщения.
- *Тесты приложений.* Тесты приложений, часто также называемые присмочными тестами, служат для проверки всего поведения приложения, обычно на его внешней границе, такой как HTTP-вызовы.
- *Регрессионные тесты.* Как и в тестах приложений, в регрессионных тестах основное внимание уделяется точному описанию предоставляемых пользователю возможностей и обеспечению их бесперебойной работы. Грань, отделяющая регрессионные тесты от тестов приложений, очень тонкая, но основное различие — в уровне детализации тестов. Например, тест приложения умеет сообщать: «Браузер может отправить POST-запрос конечной точке `people`, после чего в таблице `users` должна появиться новая запись» (сравнительно низкий уровень детализации с имитацией действий браузера). А для регрессионного теста более характерно сообщение: «После нажатия этой кнопки с введенными данными формы пользователь должен увидеть вот такой результат на этой странице» (более высокий уровень детализации с описанием фактических действий пользователей).

5.4 В Laravel тесты находятся в корневой папке `tests`. В ней расположены два файла: `TestCase.php` — базовый корневой тест, который будет расширять все ваши тесты, и `CreatesApplication.php` — трейт (импортируемый тестом `TestCase.php`), позволяющий любому классу загружать пример приложения Laravel для тестирования.

Эта папка также содержит две вложенные папки: `Features` — для тестов, охватывающих взаимодействие различных модулей друг с другом, и `Unit` — для тестов, охватывающих только одну структурную единицу кода (класс, модуль, функцию и т. д.). И в первой, и во второй папке есть файл `ExampleTest.php`, содержащий один готовый к выполнению пример теста.



Различия в тестировании до версии Laravel 5.4

В проектах, использующих версии до Laravel 5.4, каталог `tests` будет содержать только два файла: `ExampleTest.php` — пример теста и `TestCase.php` — базовый тест.

Кроме того, если ваше приложение использует одну из версий до Laravel 5.4, во всех примерах данной главы без каких-либо концептуальных изменений нужно

езде использовать немного другой синтаксис. Подробнее об этом можно узнать в документации по тестированию версии Laravel 5.3 (<http://bit.ly/2YnwDev>). Четыре самых крупных изменения включают в себя следующее.

- В версии 5.3 и ранее не нужно создавать объекты ответов. Достаточно было просто вызывать методы объекта `$this`, и класс теста сохранял ответы. То есть там, где в версии 5.4 и следующих версиях используется код `$response = $this->get('people')`, в версии 5.3 и более ранних будет код `$this->get('people')`.
- В версии 5.4 и более новых немного изменились имена утверждений для придания им большего сходства с обычными названиями утверждений PHPUnit. Так, вместо `see()` теперь используется `assertSee()`.
- В версии 5.4 и далее некоторые из методов «обхода» были выделены в пакет `browser-kit-testing`, в то время как в предыдущих версиях они были встроены в ядро.
- До версии 5.4 в Laravel не было пакета `Dusk`.

В силу того что тестирование в версиях до Laravel 5.4 несет в себе столько различий, я посвятил тестированию целую главу первого издания этой книги, доступной в виде бесплатного PDF-файла. Если вы используете Laravel 5.3 или более раннюю версию, я рекомендую пропустить данную главу и вместо нее изучить соответствующую главу из первого издания по адресу <http://bit.ly/2CNFCN1>.

Тест `ExampleTest` в вашем каталоге `Unit` содержит одно простое утверждение: `$this->assertTrue(true)`. Внутри модульных тестов почти всегда используется сравнительно простой синтаксис PHPUnit (утверждения в отношении равенства или различия значений, поиск элементов массивов, проверка логических значений и т. д.), поэтому здесь мало что требуется изучить.



Базовые сведения об утверждениях PHPUnit

Поскольку вы еще не знакомы с PHPUnit, то в большинстве случаев мы будем выполнять утверждения в объекте `$this`, используя следующий синтаксис:

```
$this->assertWHATEVER($expected, $real);
```

Так, например, в случае проверки утверждения о равенстве двух переменных следует сначала передать ожидаемый результат, а затем — фактический, выдаваемый проверяемым объектом или системой:

```
$multiplicationResult = $myCalculator->multiply(5, 3);
$this->assertEqual(15, $multiplicationResult);
```

Как видно из примера 12.1, тест `ExampleTest` из каталога `Feature` имитирует HTTP-запрос к странице с корневым адресом вашего приложения и проверяет, равен ли 200 (успешное выполнение) его HTTP-статус. Если это так, результат теста положительный, если нет — отрицательный. В отличие от обычного теста PHPUnit

мы выполняем это утверждение в объекте `TestResponse`, возвращаемом при выполнении тестового HTTP-вызова.

Пример 12.1. Тест `tests/Feature/ExampleTest.php`

```
<?php

namespace Tests\Feature;

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

class ExampleTest extends TestCase
{
    /**
     * Простейший пример теста
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

Для запуска тестов выполните команду `./vendor/bin/phpunit` в командной строке из корневой папки вашего приложения. Результаты будут выглядеть примерно так, как показано в примере 12.2.

Пример 12.2. Примерный вид результатов теста `ExampleTest`

RNPHPUnit 7.3.5 by Sebastian Bergmann and contributors.

.. 2 / 2 (100%)

Time: 139 ms, Memory: 12.00MB

OK (2 test, 2 assertions)

Примите поздравления: только что вы успешно запустили свой первый тест приложения Laravel! Две точки указывают на выполнение двух проверок. Как видно, предлагаемая «из коробки» среда тестирования включает в себя не только работающий экземпляр PHPUnit, но и полный набор инструментов для тестирования приложений, который еще может имитировать HTTP-вызовы и проверять ответы приложения. Кроме того, также можно легко воспользоваться полнофункциональным инструментом для обхода DOM-модели (см. далее врезку «Вводная информация о пакете `BrowserKit Testing`» на с. 337) и инструментом для регрессионного тестирования с полной поддержкой JavaScript (см. далее подраздел «Тестирование с использованием `Dusk`» на с. 338).

Если вы еще не знакомы с PHPUnit, посмотрим, как будет выглядеть неудачное выполнение теста. Для этого мы не будем модифицировать предыдущий тест, а создадим отдельный. Выполните команду `php artisan make:test FailingTest`. Она приведет к созданию файла `tests/Feature/FailingTest.php`. Отредактируйте внутри него метод `testExample()`, как показано в примере 12.3.

Пример 12.3. Файл `tests/Feature/FailingTest.php`, отредактированный для получения неудачного результата

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

    $response->assertStatus(301);
}
```

Данный тест выглядит так же, как предыдущий, но теперь мы проверяем на равенство другому коду состояния. Таким образом, запустим PHPUnit еще раз.



Генерирование модульных тестов

Если нужно, чтобы тест был сгенерирован в каталоге Unit, а не в Feature, следует выполнить команду с флагом `--unit`:

```
php artisan make:test SubscriptionTest --unit
```

Вуаля! На этот раз результаты теста будут выглядеть примерно так, как показано в примере 12.4.

Пример 12.4. Примерный вид результатов неудачного теста

PHPUnit 7.3.5 by Sebastian Bergmann and contributors.

```
.F.                                     3 / 3 (100%)
```

```
Time: 237 ms, Memory: 12.00MB
```

```
There was 1 failure:
```

```
1) Tests\Feature\FailingTest::testExample
Expected status code 301 but received 200.
Failed asserting that false is true.
```

```
/path-to-your-app/vendor/.../Foundation/Testing/TestResponse.php:124
/path-to-your-app/tests/Feature/FailingTest.php:20
```

```
FAILURES!
```

```
Tests: 3, Assertions: 3, Failures: 1.
```

Разберемся с происходящим. В прошлый раз у нас было только две точки, обозначающие выполнение двух тестов, но на этот раз мы видим две точки и между ними

символ F, который указывает, что один из трех выполненных здесь тестов выдал отрицательный результат.

Далее для каждой ошибки выведено имя теста (в данном случае `FailingTest::test-Example`), сообщение об ошибке (`Expected status code...`) и полная трассировка стека, чтобы мы видели, какие вызовы производились. Поскольку это был тест приложения, трассировка стека показывает лишь то, что тест был вызван с помощью класса `TestResponse`, но если бы это был модульный или функциональный тест, мы бы увидели весь стек вызовов теста.

Теперь, когда мы попробовали выполнить успешный и неудачный тесты, нужно узнать больше о среде тестирования Laravel.

Именование тестов

По умолчанию система тестирования Laravel запускает любые файлы в каталоге `tests`, имена которых заканчиваются словом `Test`. Именно поэтому в первом примере по умолчанию был запущен тест `tests/ExampleTest.php`.

В тестах PHPUnit выполняются только методы, имена которых начинаются со слова `test`, или методы блока документации `@test`. Какие методы выполняются, а какие — нет, показано в примере 12.5.

Пример 12.5. Именование методов PHPUnit

```
class NamingTest
{
    public function test_it_names_things_well()
    {
        // Выполняется
    }

    public function testItNamesThingsWell()
    {
        // Выполняется
    }

    /** @test */
    public function it_names_things_well()
    {
        // Выполняется
    }

    public function it_names_things_well()
    {
        // Не выполняется
    }
}
```

Среда тестирования

При каждом запуске приложения Laravel получает текущее имя среды, которое указывает, в среде какого типа выполняется приложение. Можно задать имя `local`, `staging`, `production` или какое-либо другое имя по вашему усмотрению. Его можно получить с помощью вызова `app()->environment()` или выполнить инструкцию `if (app()->environment('local'))` или что-то аналогичное для проверки текущей среды на соответствие переданному имени.

При запуске тестов Laravel автоматически выбирает среду `testing`. Значит, вы можете выполнять проверку `if (app()->environment('testing'))` для включения или отключения определенных поведений в среде тестирования.

Кроме того, при тестировании Laravel не загружает обычные переменные среды из файла `.env`. Если нужно задать какие-либо переменные среды для своих тестов, отредактируйте файл `phpunit.xml`, добавив в разделе `<php>` дополнительные теги `<env>` для каждой нужной вам переменной среды вида `<env name="DB_CONNECTION" value="sqlite"/>`.

ИСКЛЮЧЕНИЕ ПЕРЕМЕННЫХ СРЕДЫ ТЕСТИРОВАНИЯ ИЗ СИСТЕМЫ УПРАВЛЕНИЯ ВЕРСИЯМИ С ПОМОЩЬЮ ФАЙЛА .ENV.TESTING

Если нужно задать для своих тестов переменные среды, это можно сделать в файле `phpunit.xml` только что описанным способом. Но что, если определенные переменные среды должны отличаться в разных средах тестирования? Что, если их нужно исключить из системы управления версиями исходного кода?

К счастью, справиться с такой ситуацией просто. Сначала создайте файл `.env.testing.example`, выглядящий точно так же, как файл `Laravel .env.example`. Затем определите зависимые от среды переменные в файле `.env.testing.example`, как это делается в файле `.env.example`. Далее сделайте копию файла `.env.testing.example` и присвойте ей имя `.env.testing`. Наконец, укажите файл `.env.testing` в файле `.gitignore` непосредственно под файлом `.env` и задайте нужные значения в `.env.testing`.

В большинстве версий Laravel этот файл будет загружаться автоматически. До версии 5.2 и в некоторых второстепенных релизах до версии 5.5 этот файл может отсутствовать во фреймворке. На этот случай я написал статью в блоге по адресу <http://bit.ly/2YwnyQG>, в которой объясняется, как его добавить, если его нет во фреймворке.

Трейты тестирования

Прежде чем обсудить методы тестирования, нужно узнать о четырех трейтах проверки, которые можно включить в состав любого класса теста.

RefreshDatabase

5.5 Трейт `Illuminate\Foundation\Testing\RefreshDatabase` импортируется в начале файла каждого вновь генерируемого теста и представляет собой наиболее часто применяемый трейт миграции базы данных. Этот трейт появился в Laravel 5.5 и доступен только в проектах, использующих эту или более новую версию.

Цель этого и других трейтов для обработки базы данных — обеспечение корректной миграции таблиц базы данных перед запуском каждого теста.

Трейт `RefreshDatabase` обеспечивает это в два этапа. Во-первых, он выполняет *однократный* запуск ваших миграций в тестовой БД перед выполнением каждого теста (то есть при каждом выполнении команды `phpunit`, а не отдельного метода теста). Во-вторых, он обортывает каждый отдельный метод теста в транзакцию базы данных и выполняет ее откат после проверки.

Это означает, что ваша база данных будет мигрировать перед выполнением тестов и возвращаться к исходному состоянию после каждой проверки без необходимости повторного запуска миграции перед каждым тестом. Это самый быстрый возможный вариант. Если есть сомнения в отношении того, как вам лучше работать, не меняйте эту схему.

WithoutMiddleware

Если вы импортируете в свой класс теста трейт `Illuminate\Foundation\Testing\WithoutMiddleware`, то он будет отключать все `middleware` при выполнении любого теста этого класса. Значит, не нужно беспокоиться о `middleware` для аутентификации, CSRF-защиты или чего-либо еще, которые могут быть полезны в реальном приложении, но только отвлекают внимание в тесте.

Если нужно отключать `middleware` только для одного метода, а не всего класса теста, сделайте вызов `$this->withoutMiddleware()` в начале этого метода.

DatabaseMigrations

Если вместо трейта `RefreshDatabase` вы импортируете `Illuminate\Foundation\Testing\DatabaseMigrations`, то он будет заново запускать весь набор миграций базы данных перед каждым тестом. Laravel обеспечивает это путем выполнения команды `php artisan migrate:fresh` в методе `setUp()` перед запуском каждого теста.

DatabaseTransactions

Трейт `Illuminate\Foundation\Testing\DatabaseTransactions`, с другой стороны, ожидает, что перед запуском тестов будет выполнена надлежащая миграция базы данных. Он обортывает каждый тест в транзакцию базы данных и производит откат

этой транзакции после выполнения теста. Это означает, что после выполнения каждого теста база данных будет возвращаться в то же состояние, в котором она находилась до выполнения теста.

Простые модульные тесты

В случае простых модульных тестов вам почти не нужно использовать эти трейты. Хотя, конечно, *можно* в этих тестах осуществлять доступ к базе данных или внедрять что-либо из контейнера. В большинстве случаев модульные тесты ваших приложений будут мало зависеть от фреймворка. Простой модульный тест показан в примере 12.6.

Пример 12.6. Простой модульный тест

```
class GeometryTest extends TestCase
{
    public function test_it_calculates_area()
    {
        $square = new Square;
        $square->sideLength = 4;

        $calculator = new GeometryCalculator;

        $this->assertEquals(16, $calculator->area($square));
    }
}
```

Очевидно, что это несколько искусственный пример. Но мы тестируем здесь один класс (`GeometryCalculator`) и его единственный метод (`area()`) и делаем это, не беспокоясь обо всем приложении Laravel.

Хотя иногда модульные тесты служат для проверки чего-то, что с технической точки зрения связано с фреймворком — скажем, модели Eloquent, — даже в этом случае можно тестировать, не беспокоясь о фреймворке. Так, в примере 12.7 мы используем метод `Package::make()` вместо `Package::create()`, в результате чего объект создается и оценивается в памяти, даже не попадая в базу данных.

Пример 12.7. Более сложный модульный тест

```
class PopularityTest extends TestCase
{
    use RefreshDatabase;

    public function test_votes_matter_more_than_views()
    {
        $package1 = Package::make(['votes' => 1, 'views' => 0]);
        $package2 = Package::make(['votes' => 0, 'views' => 1]);

        $this->assertTrue($package1->popularity > $package2->popularity);
    }
}
```

В принципе, этот тест можно отнести к категории интеграционных или функциональных, поскольку данный модуль связан со всей кодовой базой Eloquent и будет взаимодействовать с БД при использовании.

Однако можно сделать такой самый важный вывод из этого примера: вы можете использовать простые тесты, которые проверяют один класс или метод, даже если тестируемые объекты связаны с фреймворком.

Следует отметить, что в большинстве случаев ваши тесты — особенно сначала — будут носить достаточно общий характер, представляя собой проверку на уровне приложения. Учитывая это, мы посвятим оставшуюся часть главы подробному рассмотрению тестирования приложений.

Как осуществляется тестирование приложений

В разделе «Основы тестирования» на с. 309 было показано, как всего несколькими строками кода можно «запрашивать» URI-адреса приложений и проверять реальный статус ответа. Но каким образом PHPUnit может запрашивать страницы словно браузер?

Класс TestCase. Любые тесты приложений должны расширять класс `TestCase` (`tests/TestCase.php`), который включен в состав Laravel по умолчанию. `TestCase` вашего приложения будет расширять абстрактный класс `Illuminate\Foundation\Testing\TestCase`, который подключает достаточно много полезного.

Эти классы `TestCase` (ваш класс и его абстрактный родительский класс) автоматически загружают экземпляр приложения `Illuminate`, предоставляя вам приложение с полностью выполненной начальной загрузкой. Они также «обновляют» состояние приложения между тестами; но при этом они не воссоздают состояние приложения *полностью*, а просто проверяют, чтобы в нем не оставалось ненужных данных.

Родительский `TestCase` также определяет систему хуков, позволяющих выполнять обратные вызовы до и после создания приложения, и импортирует ряд трейтов, предоставляющих методы для взаимодействия с каждым аспектом вашего приложения. В число импортируемых трейтов входят трейты `InteractsWithContainer`, `MakesHttpRequests` и `InteractsWithConsole`, которые предоставляют широкий спектр пользовательских утверждений и методов тестирования.

В результате в тестах приложений можно выполнять доступ к полностью загруженному экземпляру приложения и к ориентированным на тестирование приложений пользовательским утверждениям с помощью ряда простых и мощных оберток, облегчающих их использование.

Это означает, что можно написать вызов `$this->get('/')->assertStatus(200)` с уверенностью, что приложение будет вести себя так же, как если бы оно отвечало

на обычный HTTP-запрос, и что ответ будет полностью сгенерирован и затем проверен, как это делает браузер. Это довольно мощная возможность, особенно если учесть, как мало нужно для ее применения.

HTTP-тесты

Посмотрим, какие у нас есть варианты в плане написания HTTP-тестов. Вы уже видели вызов `$this->get('/')`, подробнее остановимся на том, как можно использовать этот вызов, проверять утверждения в отношении его результатов и какие еще HTTP-вызовы можно делать.

Тестирование простых страниц с помощью вызова `$this->get()` и других HTTP-вызовов

На самом базовом уровне возможности HTTP-тестирования Laravel позволяют делать простые HTTP-запросы (`GET`, `POST` и т. д.), а затем проверять простые утверждения в отношении выполняемых ими действий или их ответа.

В Laravel имеются и другие инструменты (см. во врезке «Вводная информация о пакете BrowserKit Testing» на с. 337 и в подразделе «Тестирование с использованием Dusk» на с. 338), которые позволяют проверять более сложные утверждения и взаимодействия со страницами, но начнем с базового уровня. Вы можете использовать следующие вызовы:

- ❑ `$this->get($uri, $headers = []);`
- ❑ `$this->post($uri, $data = [], $headers = []);`
- ❑ `$this->put($uri, $data = [], $headers = []);`
- ❑ `$this->patch($uri, $data = [], $headers = []);`
- ❑ `$this->delete($uri, $data = [], $headers = []);`

Эти методы составляют основу системы HTTP-тестирования. Все они принимают как минимум URI-адрес (обычно относительный) и заголовки, и все, за исключением метода `get()`, также позволяют передавать данные вместе с запросом.

Важно отметить, что все эти методы возвращают объект `$response`, представляющий HTTP-ответ. Это почти такой же объект ответа, как объект `Illuminate Response`, который мы возвращаем из контроллеров. Однако на самом деле это экземпляр класса `Illuminate\Foundation\Testing\TestResponse`, который обертывает обычный объект `Response` рядом утверждений для целей тестирования.

В примере 12.8 показаны типичное использование метода `post()` и проверка утверждения в отношении ответа.

Пример 12.8. Простейший пример тестирования с использованием метода `post()`

```
public function test_it_stores_new_packages()
{
    $response = $this->post(route('packages.store'), [
        'name' => 'The greatest package',
    ]);

    $response->assertOk();
}
```

В большинстве случаев показанный в примере 12.8 тест также будет проверять, присутствует ли запись в базе данных и отображается ли она на классификационной странице. Возможно, будет выдавать положительный результат лишь в том случае, если будет указан автора пакета и пройдена аутентификация. Но не волнуйтесь, мы еще доберемся до всего этого. Вы уже можете вызывать маршруты вашего приложения с помощью широкого набора команд и проверять утверждения в отношении ответа и состояния вашего приложения. Это не так уж мало!

Тестирование API на базе JSON с помощью вызова `$this->getJson()` и других HTTP-вызовов на базе JSON

Все описанные выше виды HTTP-тестов можно выполнять и в отношении API на базе JSON. Для этого тоже предусмотрен ряд удобных методов:

```
□ $this->getJson($uri, $headers = []);
□ $this->postJson($uri, $data = [], $headers = []);
□ $this->putJson($uri, $data = [], $headers = []);
□ $this->patchJson($uri, $data = [], $headers = []);
□ $this->deleteJson($uri, $data = [], $headers = []).
```

Эти методы работают в точности так же, как обычные методы HTTP-вызовов. Только они также добавляют специфичные для формата JSON заголовки `Accept`, `CONTENT_LENGTH` и `CONTENT_TYPE`. Как их можно использовать, показано в примере 12.9.

Пример 12.9. Простейший пример тестирования с использованием метода `postJSON()`

```
public function test_the_api_route_stores_new_packages()
{
    $response = $this->postJSON(route('api.packages.store'), [
        'name' => 'The greatest package',
    ], ['X-API-Version' => '17']);

    $response->assertOk();
}
```

Утверждения в отношении объекта `$response`

В Laravel версии 5.8 в объекте `$response` доступно 40 утверждений, поэтому для ознакомления со списком я отсылаю вас к документации по тестированию (<http://bit.ly/2H0QJqz>). Здесь же рассмотрим самые важные и часто используемые из них:

- ❑ `$response->assertOk()`. Проверяет утверждение, что код состояния ответа равен 200:

```
$response = $this->get('terms');
$response->assertOk();
```

- ❑ `$response->assertStatus($status)`. Проверяет утверждение, что код состояния ответа равен указанному коду состояния `$status`:

```
$response = $this->get('admin');
$response->assertStatus(401); // Не авторизован
```

- ❑ `$response->assertSee($text)` и `$response->assertDontSee($text)`. Проверяет утверждение, что ответ содержит/не содержит указанный текст `$text`:

```
$package = factory(Package::class)->create();
$response = $this->get(route('packages.index'));
$response->assertSee($package->name);
```

- ❑ `$response->assertJson(_array $json)`. Проверяет утверждение, что переданный массив представлен (в формате JSON) в возвращаемом JSON-сообщении:

```
$this->postJson(route('packages.store'), ['name' => 'GreatPackage2000']);
$response = $this->getJson(route('packages.index'));
$response->assertJson(['name' => 'GreatPackage2000']);
```

- ❑ `$response->assertViewHas($key, $value = null)`. Проверяет утверждение, что представление посещенной страницы содержит фрагмент данных, доступный по ключу `$key`. Опционально проверяет, что значение этой переменной равняется значению `$value`:

```
$package = factory(Package::class)->create();
$response = $this->get(route('packages.show'));
$response->assertViewHas('name', $package->name);
```

- ❑ `$response->assertSessionHas($key, $value = null)`. Проверяет утверждение о том, что сессия содержит фрагмент данных с ключом `$key`. Опционально проверяет, что значение этой переменной равняется значению `$value`:

```
$response = $this->get('beta/enable');
$response->assertSessionHas('beta-enabled', true);
```


- ❑ `$response->assertSessionHasErrors()`. Без параметров проверяет утверждение, что в специальном контейнере сессии `errors` фреймворка Laravel определена хотя бы одна ошибка. В качестве первого параметра может передаваться массив из пар «ключ/значение», описывающих определяемые ошибки, а второго — строковый формат, используемый для представления сообщений об ошибке, как показано далее:

```
// Предполагается, что маршрут "/ form" требует, чтобы было заполнено
// поле для адреса электронной почты; мы отправляем ему пустую форму,
// чтобы вызвать ошибку
$response = $this->post('form', []);
$response->assertSessionHasErrors();
$response->assertSessionHasErrors([
    'email' => 'The email field is required.',
]);
$response->assertSessionHasErrors(
    ['email' => '<p>The email field is required.</p>'],
    '<p>:message</p>'
);
```

Если вы используете именованные пакеты ошибок, как третий параметр можно передать имя пакета ошибок:

- ❑ `$response->assertCookie($name, $value = null)`. Проверяет утверждение, что ответ содержит cookie-файл с именем `$name`. Опционально проверяет, что этот файл содержит значение `$value`:

```
$response = $this->post('settings', ['dismiss-warning']);
$response->assertCookie('warning-dismiss', true);
```

- ❑ `$response->assertCookieExpired($name)`. Проверяет утверждение, что ответ содержит cookie-файл с именем `$name` и его срок действия истек:

```
$response->assertCookieExpired('warning-dismiss');
```

- ❑ `$response->assertCookieNotExpired($name)`. Проверяет утверждение, что ответ содержит cookie-файл с именем `$name` и его срок действия не истек:

```
$response->assertCookieNotExpired('warning-dismiss');
```

- ❑ `$response->assertRedirect($uri)`. Проверяет утверждение том, что запрошенный маршрут возвращает перенаправление на указанный URI-адрес:

```
$response = $this->post(route('packages.store'), [
    'email' => 'invalid'
]);
```

```
$response->assertRedirect(route('packages.create'));
```

У каждого из этих утверждений есть ряд связанных утверждений, которые здесь не были перечислены. Например, помимо утверждения `assertSessionHasErrors()`,

можно использовать `assertSessionHasNoErrors()` и `assertSessionHasErrorsIn()`. В случае утверждения `assertJson()` подходят `assertJsonCount()`, `assertJsonFragment()`, `assertJsonMissing()`, `assertJsonMissingExact()`, `assertJsonStructure()` и `assertJsonValidationErrors()`. Еще раз напомним, что вы можете ознакомиться со всем этим списком в документации.

Аутентификация ответов

Помимо прочего, с помощью тестов приложений проверяется и та часть вашего приложения, которая отвечает за аутентификацию и авторизацию. В большинстве случаев вполне достаточно возможностей цепочечного метода `actingAs()`, который принимает пользователя (или другой объект, реализующий контракт `Authenticatable`, в зависимости от настроек вашей системы). Использование показано в примере 12.10.

Пример 12.10. Простейшая аутентификация при тестировании

```
public function test_guests_cant_view_dashboard()
{
    $user = factory(User::class)->states('guest')->create();
    $response = $this->actingAs($user)->get('dashboard');
    $response->assertStatus(401); // Не авторизован
}

public function test_members_can_view_dashboard()
{
    $user = factory(User::class)->states('member')->create();
    $response = $this->actingAs($user)->get('dashboard');
    $response->assertOk();
}

public function test_members_and_guests_cant_view_statistics()
{
    $guest = factory(User::class)->states('guest')->create();
    $response = $this->actingAs($guest)->get('statistics');
    $response->assertStatus(401); // Не авторизован
    $member = factory(User::class)->states('member')->create();
    $response = $this->actingAs($member)->get('statistics');
    $response->assertStatus(401); // Не авторизован
}

public function test_admins_can_view_statistics()
{
    $user = factory(User::class)->states('admin')->create();
    $response = $this->actingAs($user)->get('statistics');
    $response->assertOk();
}
```



Авторизация с использованием состояний фабрик

Обычно при тестировании используются фабрики моделей (о которых мы уже говорили в подразделе «Фабрики моделей» на с. 120). Применение состояний фабрик моделей существенно упрощает выполнение таких задач, как создание пользователей с различными уровнями доступа.

Ряд других настроек HTTP-тестов

Если нужно задавать для своих запросов переменные сессии, можно включить в цепочку вызовов метод `withSession()`:

```
$response = $this->withSession([
    'alert-dismissed' => true,
])->get('dashboard');
```

Для определения заголовков запросов в «текущем» стиле включите в цепочку вызовов метод `withHeaders()`:

```
$response = $this->withHeaders([
    'X-THE-ANSWER' => '42',
])->get('the-restaurant-at-the-end-of-the-universe');
```

Обработка исключений в тестах приложений

Обычно исключение, выдаваемое в приложении при выполнении вами HTTP-вызовов, перехватывается обработчиком исключений Laravel и обрабатывается так же, как в обычном приложении. Поэтому тест и маршрут в примере 12.11 все равно успешно пройдут проверку, поскольку это исключение никогда не «всплывет» на уровень теста.

Пример 12.11. Данное исключение будет перехвачено обработчиком исключений Laravel, что приведет к успешному выполнению теста

```
// routes/web.php
Route::get('has-exceptions', function () {
    throw new Exception('Stop!');
});

// tests/Feature/ExceptionsTest.php
public function test_exception_in_route()
{
    $this->get('/has-exceptions');

    $this->assertTrue(true);
}
```

Это вполне уместно во многих случаях, например, когда вы ожидаете исключение валидации и хотите, чтобы фреймворк перехватывал его в обычной манере.

Но если нужно временно отключить обработчик исключений, то достаточно выполнить вызов `$this->withoutExceptionHandler()`, как показано в примере 12.12.

Пример 12.12. Временное отключение обработки исключений в одном тесте

```
// tests/Feature/ExceptionsTest.php
public function test_exception_in_route()
{
    // Здесь выдается ошибка

    $this->withoutExceptionHandler();

    $this->get('/has-exceptions');

    $this->assertTrue(true);
}
```

А если по какой-либо причине нужно снова включить обработчик исключений (например, вы отключили его в методе `setUp()` и надо вновь его включить только для одного теста), то поможет вызов `$this->withExceptionHandler()`.

Тесты базы данных

Часто при выполнении тестов требуется проверить наличие определенного эффекта в базе данных. Допустим, что нужно проверить правильность работы страницы для создания пакета. Как это лучше сделать? Выполните HTTP-вызов к конечной точке для сохранения пакета, а затем проверьте утверждение, что база данных содержит этот пакет. Так проще и безопаснее, чем проверять содержимое итоговой страницы со списком пакетов.

Для базы данных предусмотрены два основных утверждения: `$this->assertDatabaseHas()` и `$this->assertDatabaseMissing()`. Оба принимают в качестве первого параметра имя таблицы, второго — искомые данные и необязательного третьего — конкретное подключение к базе данных, которое требуется проверить.

Как их можно использовать, показано в примере 12.13.

Пример 12.13. Примеры тестов базы данных

```
public function test_create_package_page_stores_package()
{
    $this->post(route('packages.store'), [
        'name' => 'Package-a-tron',
    ]);

    $this->assertDatabaseHas('packages', ['name' => 'Package-a-tron']);
}
```

Второй параметр утверждения `assertDatabaseHas()` (искомые данные) структурирован как оператор языка SQL `WHERE` — вы передасте ключ и значение (или несколько ключей и значений), и Laravel ищет записи, соответствующие вашим ключам и значениям, в указанной таблице базы данных.

Утверждение `assertDatabaseMissing()` — инверсия утверждения `assertDatabaseHas()`.

Использование фабрик моделей в тестах

Фабрики моделей представляют собой прекрасное средство для легкого заполнения БД рандомизированными и хорошо структурированными данными для тестирования (или других целей). Вы уже видели их в действии в нескольких примерах этой главы.

Мы подробно говорили о них в подразделе «Фабрики моделей» на с. 120, поэтому вернитесь к нему, если нужно освежить знания.

Заполнение начальными данными в тестах

Если вы используете в своем приложении операции заполнения начальными данными, то можно добиться эффекта, равноценного запуску команды `php artisan db:seed`, выполнив в тесте вызов `$this->seed()`.

Вы также можете передать имя класса сидера для заполнения с использованием только этого класса:

```
$this->seed(); // Заполнение всех данных
$this->seed(UserSeeder::class); // Заполнение пользователей
```

Тестирование других систем Laravel

При тестировании систем фреймворка Laravel часто требуется приостановить их реальное действие на время проверки и вместо этого выполнить тесты поведения этих систем. Этого можно добиться путем подделки различных фасадов, в частности фасадов `Event`, `Mail` и `Notification`. О поддельных реализациях подробно рассказано в разделе «Имитирование» на с. 331, а пока рассмотрим некоторые примеры. Для каждого из рассматриваемых далее типов элементов в Laravel предусмотрен соответствующий набор утверждений, которые можно использовать после их подделки. Однако вы можете подделывать эти элементы и для отключения их действия.

Подделка событий

В качестве первого примера того, как Laravel позволяет имитировать свои внутренние системы, рассмотрим подделку событий. В некоторых случаях нужно подделать события просто для того, чтобы подавить их действие. Допустим, что ваше приложение отправляет уведомления в Slack при регистрации каждого нового пользователя. У вас есть событие «пользователь зарегистрировался», которое отправляется, когда это происходит, а также соответствующий слушатель, отправляющий уведомления, что пользователь зарегистрировался, в Slack-канал. Вам не нужно, чтобы эти уведомления отправлялись в Slack при каждом запуске тестов. Вместо этого, вероятно, требуется проверить, отправляется ли соответствующее событие, срабатывает ли слушатель и т. д. Это одна из целей подделки компонентов фреймворка Laravel в тестах — приостановить стандартное поведение и вместо этого проверить утверждения в отношении тестируемой системы.

Подавить эти события можно вызовом метода `fake()` в классе `Illuminate\Support\Facades\Event`, как показано в примере 12.14.

Пример 12.14. Подавление событий без добавления утверждений

```
public function test_controller_does_some_thing()
{
    Event::fake();

    // Вызываем контроллер и проверяем нужные вам элементы его
    // поведения, не беспокоясь об отправке им данных в Slack
}
```

После вызова метода `fake()` можно вызвать специальные утверждения в фасаде `Event::assertDispatched()` и `assertNotDispatched()`. Как их использовать, показано в примере 12.15.

Пример 12.15. Проверка утверждений в отношении событий

```
public function test_signing_up_users_notifies_slack()
{
    Event::fake();

    // Выполняем регистрацию пользователя

    Event::assertDispatched(UserJoined::class, function ($event) use ($user) {
        return $event->user->id === $user->id;
    });

    // Или выполняем регистрацию нескольких пользователей и убеждаемся в том,
    // что это событие было отправлено дважды

    Event::assertDispatched(UserJoined::class, 2);

    // Или выполняем регистрацию с неудачной валидацией и убеждаемся в том,
```

```
// что это событие не было отправлено

Event::assertNotDispatched(UserJoined::class);
}
```

Обратите внимание, что, передав утверждению `assertDispatched()` необязательное замыкание, мы убедились не только в том, что событие было отправлено, но и в том, что оно содержит определенные данные.



Метод `Event::fake()` отключает события модели Eloquent

Метод `Event::fake()` отключает события модели Eloquent. Поэтому, если важный код присутствует, например, в событии модели `creating`, проследите, чтобы создание моделей (с использованием фабрик или каким-либо иным образом) производилось до вызова метода `Event::fake()`.

Подделка фасадов Bus и Queue

С фасадом `Bus`, который представляет механизм распределения задач Laravel, можно работать так же, как с фасадом `Event`. Вы можете выполнить в нем метод `fake()` для отключения действия задач, а затем — метод `assertDispatched()` или `assertNotDispatched()`.

Фасад `Queue` представляет используемый Laravel механизм распределения задач после их помещения в очередь. Он предлагает методы `assertedPushed()`, `assertPushedOn()` и `assertNotPushed()`.

Как подделывать оба фасада, показано в примере 12.16.

Пример 12.16. Подделка задач, не помещенных и помещенных в очередь

```
public function test_popularity_is_calculated()
{
    Bus::fake();

    // Синхронизируем данные пакета...

    // Убеждаемся в том, что задача была распределена
    Bus::assertDispatched(
        CalculatePopularity::class,
        function ($job) use ($package) {
            return $job->package->id === $package->id;
        }
    );

    // Убеждаемся в том, что задача не была распределена
    Bus::assertNotDispatched(DestroyPopularityMaybe::class);
}

public function test_popularity_calculation_is_queued()
```

```

{
    Queue::fake();

    // Синхронизируем данные пакета...

    // Убеждаемся в том, что задача была помещена в очередь
    Queue::assertPushed(CalculatePopularity::class, function ($job) use ($package)
    {
        return $job->package->id === $package->id;
    });

    // Убеждаемся в том, что задача была помещена в указанную очередь "popularity"
    Queue::assertPushedOn('popularity', CalculatePopularity::class);

    // Убеждаемся в том, что задача была помещена в очередь дважды
    Queue::assertPushed(CalculatePopularity::class, 2);

    // Убеждаемся в том, что задача не была помещена в очередь
    Queue::assertNotPushed(DestroyPopularityMaybe::class);
}

```

Подделка фасада Mail

Для подделки фасада Mail есть четыре метода: `assertSent()`, `assertNotSent()`, `assertQueued()` и `assertNotQueued()`. Используйте методы `Queued`, если ваша почта организована в виде очереди, и `Sent` — в противном случае.

Как и в случае метода `assertDispatched()`, в качестве первого параметра передается имя отправляемого сообщения, а необязательного второго — количество выполняемых операций отправки сообщения или замыкание, проверяющее корректность содержащихся в сообщении данных. Пример 12.17 демонстрирует некоторые из них в действии.

Пример 12.17. Проверка утверждений в отношении электронной почты

```

public function test_package_authors_receive_launch_emails()
{
    Mail::fake();

    // Впервые делаем пакет публичным...

    // Убеждаемся, что сообщение было отправлено
    // на указанный адрес электронной почты
    Mail::assertSent(PackageLaunched::class, function ($mail) use ($package) {
        return $mail->package->id === $package->id;
    });

    // Убеждаемся, что сообщение было отправлено
    // на указанные адреса электронной почты
    Mail::assertSent(PackageLaunched::class, function ($mail) use ($package) {
        return $mail->hasTo($package->author->email) &&
            $mail->hasCc($package->collaborators) &&

```



```
        $mail->hasBcc('admin@novapackages.com');
    });

    // Или запускаем два пакета...

    // Убеждаемся в том, что сообщение было отправлено дважды
    Mail::assertSent(PackageLaunched::class, 2);

    // Убеждаемся в том, что сообщение не было отправлено
    Mail::assertNotSent(PackageLaunchFailed::class);
}
```

Все методы для проверки сообщений в отношении указанных получателей (`hasTo()`, `hasCc()` и `hasBcc()`) могут принимать либо один адрес электронной почты, либо массив или коллекцию адресов.

Подделка фасада Notification

При подделке фасада `Notification` можно использовать два метода: `assertSentTo()` и `assertNothingSent()`.

В отличие от фасада `Mail` при этом не нужно вручную проверять в замыкании, кому было отправлено уведомление. Вместо этого само утверждение требует, чтобы в качестве первого параметра ему передавался либо один уведомляемый объект, либо массив или коллекция таких объектов. Проверить что-либо в отношении самого уведомления можно лишь после того, как вы передадите утверждению получателя уведомления.

В качестве второго параметра передается имя класса уведомления, а необязательного третьего — замыкание, определяющее дополнительные требования к уведомлению. Как это делается, показано в примере 12.18.

Пример 12.18. Подделка фасада Notification

```
public function test_users_are_notified_of_new_package_ratings()
{
    Notification::fake();

    // Производим оценку пакета...

    // Убеждаемся в том, что автор был уведомлен
    Notification::assertSentTo(
        $package->author,
        PackageRatingReceived::class,
        function ($notification, $channels) use ($package) {
            return $notification->package->id === $package->id;
        }
    );

    // Убеждаемся в том, что уведомление было отправлено
    // указанным пользователям
}
```

```

Notification::assertSentTo(
    [$package->collaborators], PackageRatingReceived::class
);

// Или производим оценку двойного пакета...

// Убеждаемся в том, что уведомление не было отправлено
Notification::assertNotSentTo(
    [$package->author], PackageRatingReceived::class
);
}

```

Иногда нужно убедиться, что выбор каналов производится должным образом и уведомления отправляются по нужным каналам (пример 12.19).

Пример 12.19. Тестирование каналов уведомлений

```

public function test_users_are_notified_by_their_preferred_channel()
{
    Notification::fake();

    $user = factory(User::class)->create(['slack_preferred' => true]);

    // Производим оценку пакета...

    // Убеждаемся в том, что автор был уведомлен через Slack
    Notification::assertSentTo(
        $user,
        PackageRatingReceived::class,
        function ($notification, $channels) use ($package) {
            return $notification->package->id === $package->id
                && in_array('slack', $channels);
        }
    );
}

```

Подделка фасада Storage

Тестирование работы с файлами может быть чрезвычайно сложным. При использовании традиционных подходов часто требуется перемещать файлы для проверки в специальные каталоги и обрабатывать сложное форматирование входных и выходных данных форм.

Если вы используете фасад **Storage** фреймворка Laravel, то тестирование загрузки файлов на сервер или других аспектов сохранения упрощается. Это иллюстрирует пример 12.20.

Пример 12.20. Тестирование хранилища и загрузки файлов на сервер с подделкой хранилища

```

public function test_package_screenshot_upload()
{
    Storage::fake('screenshots');
}

```

```
// Загружаем на сервер поддельное изображение
$response = $this->postJson('screenshots', [
    'screenshot' => UploadedFile::fake()->image('screenshot.jpg'),
]);

// Убеждаемся в том, что файл был сохранен
Storage::disk('screenshots')->assertExists('screenshot.jpg');

// Или убеждаемся в том, что файл не существует
Storage::disk('screenshots')->assertMissing('missing.jpg');
}
```

Имитирование

Имитации (а также их братья по оружию — шпионы, заглушки, макеты, поддельные реализации и другие подобные инструменты) широко применяются в тестировании. Мы видели несколько примеров применения поддельных реализаций в предыдущем разделе. Можно сказать, что вы вряд ли сможете должным образом протестировать приложение любого размера, совсем не имитируя какие-либо компоненты.

Поэтому кратко остановимся на том, что представляет собой имитирование в Laravel и каким образом применяется библиотека для имитирования Mockery.

Вводная информация об имитировании

Имитации и подобные инструменты, по сути, позволяют создать объект, который в определенной мере имитирует реальный класс, но для целей тестирования не является реальным. Иногда это нужно, чтобы внедрить в тест класс, который в его реальном виде очень сложно инстанцируется или взаимодействует с внешним сервисом.

Как вы увидите в примерах далее, Laravel поощряет разработчика для работы с реальным приложением, не впадая в большую зависимость от использования имитаций. В то же время им находится применение, и поэтому Laravel по умолчанию предлагает библиотеку для имитирования Mockery, а многие из его основных сервисов предлагают методы подделки.

Вводная информация о Mockery

Mockery позволяет быстро и легко создавать имитации любого класса PHP, используемого в вашем приложении. Представим, что есть класс, зависящий от клиента Slack, но вы не хотите, чтобы эти вызовы действительно отправлялись в Slack. Mockery помогает создать фиктивный клиент Slack и применить его в своих тестах, как показано в примере 12.21.

Пример 12.21. Использование Mockery в Laravel

```
// app/SlackClient.php
class SlackClient
{
    // ...

    public function send($message, $channel)
    {
        // Отправляет реальное сообщение в Slack
    }
}

// app/Notifier.php
class Notifier
{
    private $slack;

    public function __construct(SlackClient $slack)
    {
        $this->slack = $slack;
    }

    public function notifyAdmins($message)
    {
        $this->slack->send($message, 'admins');
    }
}

// tests/Unit/NotifierTest.php
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class)->shouldIgnoreMissing();

    $notifier = new Notifier($slackMock);
    $notifier->notifyAdmins('Test message');
}
```

Здесь задействовано много элементов. Если мы разберемся, что представляет собой каждый из них, легко уловить общий смысл. У нас есть тестируемый класс с именем `Notifier`. У него зависимость с именем `SlackClient`, которая делает то, что не следует делать при выполнении тестов: отправляет реальные уведомления в Slack. Поэтому нужно симитировать эту зависимость.

Используя `Mockery`, мы создаем имитацию класса `SlackClient`. Если нам вообще неважно, что происходит с этим классом, то есть достаточно, чтобы он просто присутствовал, чтобы наши тесты не выдавали ошибок, можно воспользоваться методом `shouldIgnoreMissing()`:

```
$slackMock = Mockery::mock(SlackClient::class)->shouldIgnoreMissing();
```

Независимо от того, что будет вызывать объект `Notifier` в объекте `$slackMock`, последний будет просто принимать эти вызовы и возвращать значение `null`.

Однако взгляните на метод `test_notifier_notifies_admins()`. Пока он не выполняет никакого реального тестирования.

Мы могли бы просто оставить здесь вызов метода `shouldReceiveMissing()`, лишь добавив ниже несколько утверждений. Именно так обычно используется метод `shouldReceiveMissing()`, делающий объект поддельной реализацией (заглушкой).

Но если нужно проверить, был ли произведен реальный вызов метода `send()` объекта `SlackClient`? В таком случае вместо `shouldReceiveMissing()` следует воспользоваться другими методами `shouldReceive*` (пример 12.22).

Пример 12.22. Использование метода `shouldReceive()` в имитации `Mockery`

```
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class);
    $slackMock->shouldReceive('send')->once();

    $notifier = new Notifier($slackMock);
    $notifier->notifyAdmins('Test message');
}
```

Суть вызова `shouldReceive('send')->once()` можно выразить следующими словами: «Убедиться в том, что метод `send()` объекта `$slackMock` будет вызван только один раз». Так мы убеждаемся, что при вызове метода `notifyAdmins()` объект `Notifier` будет делать однократный вызов `send()` в `SlackClient`.

Мы также могли бы написать здесь что-то вроде `shouldReceive('send')->times(3)` или `shouldReceive('send')->never()`. С помощью метода `with()` можно указать, какой параметр должен передаваться вместе с вызовом метода `send()`, а с помощью `andReturn()` можно определить возвращаемое значение:

```
$slackMock->shouldReceive('send')->with('Hello, world!')->andReturn(true);
```

Что, если мы захотим использовать IoC-контейнер для разрешения нашего экземпляра класса `Notifier`? Это может быть полезным, если класс `Notifier` имеет несколько других зависимостей, которые не требуется имитировать.

Что ж, это вполне осуществимо! Нужно просто вызвать метод `instance()` в контейнере, как показано в примере 12.23, тем самым дав Laravel указание предоставлять экземпляр нашей имитации всем запрашивающим ее классам (`Notifier` в данном примере).

Пример 12.23. Привязка экземпляра `Mockery` к контейнеру

```
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class);
    $slackMock->shouldReceive('send')->once();

    app()->instance(SlackClient::class, $slackMock);
}
```

```

    $notifier = app(Notifier::class);
    $notifier->notifyAdmins('Test message');
}

```

В Laravel 5.8 и следующих можно использовать удобный сокращенный способ создания и привязки экземпляра Mockery к контейнеру (пример 12.24).

Пример 12.24. Упрощенный способ привязки экземпляров Mockery к контейнеру в Laravel 5.8 и более новых версиях

```

$this->mock(SlackClient::class, function ($mock) {
    $mock->shouldReceive('send')->once();
});

```

Это далеко не все, что позволяет делать Mockery, — можно использовать шпионы, частичные шпионы и многое другое. Углубленное изучение работы с Mockery выходит за рамки данной книги. Однако я советую подробнее узнать, что собой представляет и как работает эта библиотека, ознакомившись с ее документацией по адресу <http://bit.ly/2Op4yyN>.

Подделка других фасадов

Библиотека Mockery предлагает еще одну интересную возможность: вызывать ее методы (например, `shouldReceive()`) в любых фасадах приложения.

Допустим, у нас есть метод контроллера, который использует фасад, не относящийся к числу рассмотренных подделываемых систем. Нам нужно протестировать этот метод контроллера и убедиться, что производится вызов определенного фасада.

Справиться с этой задачей достаточно просто: вызвать методы Mockery в соответствующем фасаде, как показано в примере 12.25.

Пример 12.25. Имитирование фасада

```

// PeopleController
public function index()
{
    return Cache::remember('people', function () {
        return Person::all();
    });
}

// PeopleTest
public function test_all_people_route_should_be_cached()
{
    $person = factory(Person::class)->create();

    Cache::shouldReceive('remember')
        ->once()
        ->andReturn(collect([$person]));

    $this->get('people')->assertJsonFragment(['name' => $person->name]);
}

```

Мы можем применять методы вроде `shouldReceive()` в фасадах так же, как они используются в объекте `Mockery`.

Можно сделать свои фасады шпионами, разместив утверждения в конце и используя метод `shouldReceive()` вместо `shouldReceive()`, как показано в примере 12.26.

Пример 12.26. Фасады-шпионы

```
public function test_package_should_be_cached_after_visit()
{
    Cache::spy();

    $package = factory(Package::class)->create();

    $this->get(route('packages.show', [$package->id]));

    Cache::shouldReceive('put')
        ->once()
        ->with('packages.' . $package->id, $package->toArray());
}
```

Тестирование команд Artisan

В этой главе мы рассмотрели много инструментов из арсенала средств тестирования Laravel и близки к финишу. Осталось поговорить лишь о том, как выполняется проверка браузера и команд Artisan.

5.6 Если вы используете версии до Laravel 5.7, то лучший способ тестирования команд Artisan — вызвать их с помощью метода `$this->artisan($commandName, $parameters)` и проверить их работу, как показано в примере 12.27.

Пример 12.27. Простые тесты команд Artisan

```
public function test_promote_console_command_promotes_user()
{
    $user = factory(User::class)->create();

    $this->artisan('user:promote', ['userId' => $user->id]);

    $this->assertTrue($user->isPromoted());
}
```

Можно проверять утверждения в отношении получаемого от команды Artisan кода возврата, как показано в примере 12.28.

Пример 12.28. Проверка вручную утверждений в отношении кодов возврата команд Artisan

```
$code = $this->artisan('do:thing', ['--flagOfSomeSort' => true]);
$this->assertEquals(0, $code); // 0 означает "команда не возвращает ошибки"
```

5.7 Утверждения в отношении синтаксиса команд Artisan. Если вы работаете с Laravel 5.7 и более новыми версиями, можно пристыковать к своему вызову `$this->artisan()` три новых метода: `expectsQuestion()`, `expectsOutput()` и `assertExitCode()`. Методы `expects*` будут работать совместно с любыми интерактивными инструкциями, включая `confirm()` и `anticipate()`, а `assertExitCode()` — псевдоним для кода в примере 12.28.

Применение методов см. в примере 12.29.

Пример 12.29. Простейшие тесты в отношении «ожиданий» команд Artisan

```
// routes/console.php
Artisan::command('make:post {--expanded}', function () {
    $title = $this->ask('What is the post title?');
    $this->comment('Creating at ' . str_slug($title) . '.md');

    $category = $this->choice('What category?', ['technology', 'construction'], 0);

    // Создание сообщения

    $this->comment('Post created');
});

// Файл теста
public function test_make_post_console_commands_performs_as_expected()
{
    $this->artisan('make:post', ['--expanded' => true])
        ->expectsQuestion('What is the post title?', 'My Best Post Now')
        ->expectsOutput('Creating at my-best-post-now.md')
        ->expectsQuestion('What category?', 'construction')
        ->expectsOutput('Post created')
        ->assertExitCode(0);
}
```

В качестве первого параметра методу `expectsQuestion()` передается ожидаемый текст вопроса, а второго — текст ответа на вопрос. `expectsOutput()` просто проверяет, возвращается ли переданная ему строка.

Браузерные тесты

Вот мы и дошли до тестов браузера! Они позволяют реально взаимодействовать с DOM-моделью ваших страниц: в тестах браузера можно нажимать кнопки, заполнять и отправлять формы, а в случае использования Dusk даже взаимодействовать с JavaScript.

Laravel предлагает два отдельных инструмента для тестирования браузера: BrowserKit Testing и Dusk. Активно поддерживается только Dusk. Пакет BrowserKit

Testing, похоже, уже перешел в разряд «граждан второго сорта», но на момент написания этой книги еще доступен в GitHub и по-прежнему работает.

Выбор инструмента

Для проверки браузера рекомендую по возможности использовать базовые инструменты тестирования приложений (которые мы рассмотрели). Если в вашем приложении нет JavaScript и нужно протестировать реальные манипуляции с DOM-моделью или элементы пользовательского интерфейса форм, поможет пакет BrowserKit. Если вы разрабатываете приложение, в котором активно применяется JavaScript, то лучше использовать пакет Dusk, о котором мы поговорим далее.

Иногда нужен набор тестов на базе JavaScript, использующий что-то вроде пакетов Jest и vue-test-utils (рассмотрение которых выходит за рамки данной книги). Они очень полезны для тестирования компонентов Vue, а функция снимков состояния пакета Jest упрощает синхронизацию тестовых данных API и клиентской части. Чтобы узнать об этом более подробно, ознакомьтесь с публикацией в блоге «Начало работы» Калеба Порцио (<http://bit.ly/2OuchSI>) и посмотрите видео с выступлением Саманты Гейтц на конференции Laracon в 2018 году (<http://bit.ly/2UY8nNS>).

Если у вас какой-либо другой JavaScript-фреймворк, то в мире Laravel пока нет рекомендованных решений для тестирования клиентской части. В то же время достаточно обширное сообщество разработчиков, использующих React, в основном предпочитает пакеты Jest и Enzyme.

ВВОДНАЯ ИНФОРМАЦИЯ О ПАКЕТЕ BROWSERKIT TESTING

Пакет BrowserKit Testing — это код тестов приложений для версий до Laravel 5.4, выделенный в виде отдельного пакета. BrowserKit представляет собой компонент, выполняющий синтаксический разбор DOM-модели и позволяющий вам «выбирать» элементы DOM-модели и взаимодействовать с ними. Это отлично подходит для таких простых взаимодействий со страницей, как щелчок на ссылке или заполнение форм, но не работает для JavaScript-кода.

Хотя пакет Browserkit Testing не был отнесен к числу не рекомендуемых, он совсем не упоминается в документации и производит явное впечатление нерекондуемого устаревшего кода. По этой причине, а также в силу надежности встроенного набора инструментов для тестирования приложений я не буду описывать работу с этим пакетом. Это подробно рассмотрено в первом издании этой книги, поэтому, если вы собираетесь использовать Browserkit Testing, ознакомьтесь с бесплатным PDF-файлом, содержащим посвященную тестированию главу из первого издания (<http://bit.ly/2CNFCN1>).

Тестирование с использованием Dusk

Dusk — это инструмент Laravel (устанавливаемый как пакет утилиты Composer), который позволяет легко писать инструкции в стиле Selenium, обеспечивающие взаимодействие браузера на основе ChromeDriver с вашим приложением. В отличие от большинства других инструментов на базе Selenium, Dusk предлагает простой API, позволяющий писать код для взаимодействия с ним вручную. Взгляните на следующий пример:

```
$this->browse(function ($browser) {
    $browser->visit('/register')
        ->type('email', 'test@example.com')
        ->type('password', 'secret')
        ->press('Sign Up')
        ->assertPathIs('/dashboard');
});
```

При использовании Dusk реальный браузер интерпретирует все ваше приложение и взаимодействует с ним. Значит, вы можете тестировать сложные взаимодействия, выполняемые с помощью JavaScript-кода, и получать снимки состояния при возникновении ошибок. В то же время это означает, что тесты работают немного медленнее и в них легче ошибиться, чем в случае использования базовых средств Laravel для тестирования приложений.

Лично я нахожу Dusk очень полезным в качестве инструмента для регрессионного тестирования — при этом он работает лучше, чем такие инструменты, как Selenium. Я не использую его для всех разработок на основе тестирования, а только убеждаюсь с его помощью в отсутствии нарушений во взаимодействии с пользователем (то есть регресса) по мере разработки. Можно рассматривать это как написание тестов для пользовательского интерфейса после его создания.

Поскольку пакет Dusk снабжен качественной документацией (<http://bit.ly/2JF0POY>), я не буду вдаваться в дальнейшие подробности, а просто продемонстрирую основы работы с ним.

Установка Dusk

Чтобы установить Dusk, выполните следующие две команды:

```
composer require --dev laravel/dusk
php artisan dusk:install
```

Затем отредактируйте файл `.env`, присвоив переменной `APP_URL` тот же URL-адрес, который вы используете для просмотра своего сайта в локальном браузере, — что-то наподобие `http://mysite.test`.

Для запуска тестов Dusk выполните команду `php artisan dusk`. При этом можно передать все те же параметры, которые вы использовали совместно с PHPUnit (например, `php artisan dusk --filter=my_best_test`).

Написание тестов Dusk

Для того чтобы сгенерировать новый тест Dusk, используйте команду следующего вида:

```
php artisan dusk:make RatingTest
```

Данный тест будет сохранен в файле `tests/Browser/RatingTest.php`.



Настройка переменных среды для Dusk

Можно настроить переменные среды для Dusk, создав новый файл с именем `.env.dusk.local` (в зависимости от среды вместо типа `local` в имени может быть указан другой тип, например `staging`).

Чтобы написать свои тесты Dusk, представьте, что даете одному или нескольким браузерам указание перейти по адресу вашего приложения и выполнить определенные действия. Как при этом будет выглядеть синтаксис, показано в примере 12.30.

Пример 12.30. Простой тест Dusk

```
public function testBasicExample()
{
    $user = factory(User::class)->create();

    $this->browse(function ($browser) use ($user) {
        $browser->visit('login')
            ->type('email', $user->email)
            ->type('password', 'secret')
            ->press('Login')
            ->assertPathIs('/home');
    });
}
```

Метод `$this->browse()` создает объект браузера, который передается в замыкание, после чего внутри замыкания вы указываете браузеру, какие действия следует выполнить.

В отличие от других инструментов Laravel для тестирования приложений, которые имитируют поведение ваших форм, Dusk производит реальную интерпретацию работы браузера, отправляя ему события, соответствующие вводу определенных слов или нажатию кнопок. Это реальный браузер, и Dusk управляет им в полной мере.

Вы также можете «опрашивать» более одного браузера, добавляя соответствующие параметры в замыкание, что позволит вам проверить, как много пользователей могут взаимодействовать с сайтом (например, по чату). Взгляните на пример 12.31, представляющий собой выдержку из документации.

Пример 12.31. Проверка с помощью Dusk нескольких браузеров

```
$this->browse(function ($first, $second) {  
    $first->loginAs(User::find(1))  
        ->visit('home')  
        ->waitForText('Message');  
  
    $second->loginAs(User::find(2))  
        ->visit('home')  
        ->waitForText('Message')  
        ->type('message', 'Hey Taylor')  
        ->press('Send');  
  
    $first->waitForText('Hey Taylor')  
        ->assertSee('Jeffrey Way');  
});
```

Можно использовать весьма обширный набор действий и утверждений, которые мы не будем рассматривать здесь (обратитесь к документации), но лучше изучим некоторые инструменты из арсенала пакета Dusk.

Аутентификация и базы данных

Как видно из примера 12.31, в тестах Dusk используется несколько иной синтаксис аутентификации по сравнению с другими средствами Laravel для проверки приложений: `$browser->loginAs($user)`.



Не используйте трейт RefreshDatabase совместно с Dusk

Ни в коем случае не используйте совместно с Dusk трейт RefreshDatabase! Лучше DatabaseMigrations, поскольку транзакции, используемые в RefreshDatabase, не сохраняются при переходе от одного запроса к другому.

Взаимодействие со страницей

Если вам приходилось работать с jQuery, то взаимодействие со страницей с использованием Dusk не вызовет вопросов. В примере 12.32 показаны типичные шаблоны выбора различных элементов с помощью Dusk.

Пример 12.32. Выбор элементов с помощью Dusk

```
<-- Template -->  
<div class="search"><input><button id="search-button"></button></div>  
<button dusk="expand-nav"></button>  
  
// Тесты Dusk  
// Вариант 1: синтаксис в стиле jQuery  
$browser->click('.search button');  
$browser->click('#search-button');  
  
// Вариант 2, рекомендуемый: синтаксис вида dusk="нужный-селектор"  
$browser->click('@expand-nav');
```

Добавив атрибут `dusk` в элементы страницы, можно напрямую ссылаться на них. И на это никак не повлияют последующие изменения в способе отображения или макете страницы — когда какой-либо метод запрашивает селектор, ему передается знак `@`, за которым следует содержимое соответствующего атрибута `dusk`.

Рассмотрим некоторые из методов, которые можно вызывать в объекте `$browser`. Для работы с текстом и значениями атрибутов используются следующие методы.

- ❑ `value($selector, $value = null)`. Если передан только один параметр, возвращает значение указанного элемента текстового ввода; если передан второй — устанавливает значение элемента ввода.
- ❑ `text($selector)`. Получает текстовое содержимое незаполняемого элемента, такого как `<div>` или `<span>`.
- ❑ `attribute($selector, $attributeName)`. Возвращает значение конкретного атрибута элемента, соответствующего селектору `$selector`.

Для работы с формами и файлами используются следующие методы.

- ❑ `type($selector, $valueToType)`. Аналогичен методу `value()`, но вместо установки значения выполняет реальный вывод символов.



Порядок сопоставления селекторов Dusk

В случае методов, которые, подобно методу `type()`, выбирают некоторый элемент ввода, Dusk сначала пытается сопоставить селектор Dusk или CSS, затем — элемент ввода с указанным именем и наконец, — элемент `<textarea>` с указанным именем.

- ❑ `select($selector, $optionValue)`. Выбирает вариант со значением `$optionValue` в раскрывающемся меню, выбираемом селектором `$selector`.
- ❑ `check($selector)` и `unchecked($selector)`. Устанавливает или снимает флажок, выбираемый селектором `$selector`.
- ❑ `radio($selector, $optionValue)`. Выбирает вариант со значением `$optionValue` в группе переключателей, задаваемой селектором `$selector`.
- ❑ `attach($selector, $filePath)`. Прикрепляет файл, расположенный по пути `$filePath`, к файловому вводу, выбираемому `$selector`.

Методы для ввода клавиатуры и мыши.

- ❑ `clickLink($selector)`. Производит переход по предоставленной текстовой ссылке.
- ❑ `click($selector)` и `mouseover($selector)`. Запускает щелчок кнопкой мыши или событие наведения указателя мыши в объекте `$selector`.
- ❑ `drag($selectorToDrag, $selectorToDragTo)`. Перетаскивает один элемент к другому.

- ❑ `dragLeft()`, `dragRight()`, `dragUp()`, `dragDown()`. Перетаскивает влево, вправо, вверх или вниз элемент, выбираемый селектором, переданным в качестве первого параметра, на количество пикселей, отправленное как второй параметр.
- ❑ `keys($selector, $instructions)`. Отправляет события нажатия клавиш в контексте `$selector` в соответствии с инструкциями `$instructions`. При этом можно комбинировать модификаторы с текстом:

```
$browser->keys('selector', 'this is ', ['{shift}', 'great']);
```

Данный код выведет строку `this is GREAT`. Добавив массив в список выводимых элементов, мы скомбинировали модификаторы (обернутые в скобки `{}`) с непосредственным текстом. С полным списком доступных для использования модификаторов можно ознакомиться в источнике Facebook WebDriver по адресу <http://bit.ly/2uB5APj>.

Если нужно отправить странице последовательность клавиш (например, чтобы активизировать их сочетание), то можно выбрать с помощью селектора верхний уровень приложения или страницы. Например, если это приложение Vue, а верхний уровень — элемент `<div>` с идентификатором `app`, код будет выглядеть так:

```
$browser->keys('#app', ['{command}', '/']);
```

Ожидание

Поскольку Dusk взаимодействует с JavaScript и управляет реальным браузером, мы должны коснуться и таких понятий, как время выполнения и ожидания. Dusk предлагает несколько методов для обеспечения надлежащей временной синхронизации тестов. Некоторые из них используются для взаимодействия с элементами страницы, которые специально сделаны медленными или работающими с задержкой, в то время как другие также можно применять для обхода времени инициализации используемых компонентов. Доступны следующие методы.

- ❑ `pause($milliseconds)`. Приостанавливает выполнение тестов Dusk на указанное количество миллисекунд. Это самый простой способ ожидания; он заставляет любые отправляемые браузеру последующие команды выжидать указанное количество времени перед выполнением действий.

Этот и другие методы ожидания можно использовать внутри цепочки утверждений, как показано ниже:

```
$browser->click('chat')
->pause(500)
->assertSee('How can we help?');
```

- ❑ `waitFor($selector, $maxSeconds = null)` и `waitForMissing($selector, $maxSeconds = null)`. Ожидает до момента, когда на странице появится (`waitFor()`) или исчезнет (`waitForMissing()`) указанный элемент. Дополни-

тельно может отсчитываться время ожидания в секундах, переданное в качестве необязательного второго параметра:

```
$browser->waitFor('@chat', 5);
$browser->waitUntilMissing('@loading', 5);
```

- ❑ **whenAvailable(\$selector, \$callback)**. Аналогичен методу **waitFor()**, но принимает в качестве второго параметра замыкание, которое определяет, какое действие следует выполнять после того, как указанный элемент станет доступным:

```
$browser->whenAvailable('@chat', function ($chat) {
    $chat->assertSee('How can we help you?');
});
```

- ❑ **waitForText(\$text, \$maxSeconds = null)**. Ожидает появления текста на странице. Дополнительно может отсчитываться время ожидания в секундах, переданное в качестве необязательного второго параметра:

```
$browser->waitForText('Your purchase has been completed.', 5);
```

- ❑ **waitForLink(\$linkText, \$maxSeconds = null)**. Ожидает появления ссылки с указанным текстом. Дополнительно может отсчитываться время ожидания в секундах, переданное в качестве необязательного второго параметра:

```
$browser->waitForLink('Clear these results', 2);
```

- ❑ **waitForLocation(\$path)**. Ожидает, когда URL-адрес страницы совпадет с указанным путем:

```
$browser->waitForLocation('auth/login');
```

- ❑ **waitForRoute(\$routeName)**. Ожидает, когда URL-адрес страницы совпадет с URL-адресом указанного маршрута:

```
$browser->waitForRoute('packages.show', [$package->id]);
```

- ❑ **waitForReload()**. Ожидает перезагрузки страницы.

- ❑ **waitUntil(\$expression)**. Ожидает, пока указанное выражение JavaScript не станет истинным:

```
$browser->waitUntil('App.packages.length > 0', 7);
```

Другие утверждения

Как упоминалось выше, с помощью Dusk можно проверить множество различных утверждений в отношении своего приложения. Вот некоторые из них, используемые мной чаще всего, — полный список см. в документации по Dusk (<https://laravel.com/docs/dusk>):

- ❑ **assertTitleContains(\$text);**
- ❑ **assertQueryStringHas(\$keyName);**

```

❑ assertHasCookie($cookieName);
❑ assertSourceHas($htmlSourceCode);
❑ assertChecked($selector);
❑ assertSelectHasOption($selectorForSelect, $optionValue);
❑ assertVisible($selector);
❑ assertFocused();
❑ assertVue($dataLocation, $dataValue, $selector).

```

Другие организационные структуры

Все, что мы рассмотрели выше, позволяет тестировать отдельные элементы на наших страницах. Однако Dusk также часто используется для тестирования одностраничных и более сложных приложений. Следовательно, нам потребуется заключить наши утверждения в некоторые организационные структуры.

Первой организационной структурой, с которой мы столкнулись, был атрибут `dusk` (который, например, в элементе `<div dusk="abc">` создает селектор с именем `@abc`, позволяющий обратиться к этому элементу позднее) и замыкания, которые мы можем использовать для обертывания определенных частей нашего кода (скажем, с помощью метода `whenAvailable()`).

Помимо этого, Dusk предлагает еще два типа организационных структур: страницы и компоненты. Начнем со страниц.

Страницы. Страница — это генерируемый вами класс, который включает в себя два элемента функциональности. Первый — URL-адрес и утверждения, определяющие, какую страницу вашего приложения следует сопоставить с данной страницей Dusk. Второй — сокращенный способ записи, наподобие уже применявшегося выше встроенного кода (вспомните селектор `@abc`, генерируемый атрибутом `dusk="abc"` в нашем HTML-коде), но только для данной страницы и без необходимости редактировать наш HTML-код.

Предположим, что в нашем приложении есть страница для создания пакета. Сгенерировать соответствующую страницу Dusk можно следующим образом:

```
php artisan dusk:page CreatePackage
```

Пример 12.33 демонстрирует сгенерированный нами класс.

Пример 12.33. Сгенерированная страница Dusk

```
<?php
```

```
namespace Tests\Browser\Pages;
```



```
use Laravel\Dusk\Browser;

class CreatePackage extends Page
{
    /**
     * Получаем URL-адрес страницы
     *
     * @return string
     */
    public function url()
    {
        return '/';
    }

    /**
     * Убеждаемся в том, что браузер перешел на страницу
     *
     * @param Browser $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertPathIs($this->url());
    }

    /**
     * Получаем псевдонимы элементов для страницы
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@element' => '#selector',
        ];
    }
}
```

Метод `url()` определяет адрес, по которому Dusk будет искать страницу. `assert()` позволяет выполнить дополнительные проверки, чтобы убедиться, что вы находитесь на нужной странице; а метод `elements()` предоставляет псевдонимы для селекторов в стиле `@element`.

Немного отредактируем нашу страницу для создания пакета, как показано в примере 12.34.

Пример 12.34. Простая страница Dusk для создания пакета

```
class CreatePackage extends Page
{
    public function url()
    {
```

```

        return '/packages/create';
    }

    public function assert(Browser $browser)
    {
        $browser->assertTitleContains('Create Package');
        $browser->assertPathIs($this->url());
    }

    public function elements()
    {
        return [
            '@title' => 'input[name=title]',
            '@instructions' => 'textarea[name=instructions]',
        ];
    }
}

```

Теперь, когда у нас есть работающая страница, мы можем перейти к ней и получить доступ к определенным на ней элементам:

```

// Внутри теста
$browser->visit(new Tests\Browser\Pages\CreatePackage)
    ->type('@title', 'My package title');

```

Страницы применяются для определения часто используемых действий, которые вам нужно выполнять в тестах. То есть вы создаете своего рода макросы для Dusk. При этом определяете метод на своей странице, а затем вызываете его из своего кода, как показано в примере 12.35.

Пример 12.35. Определение и использование собственного метода страницы

```

class CreatePackage extends Page
{
    // ... url(), assert(), elements()

    public function fillBasicFields(Browser $browser, $packageTitle = 'Best package')
    {
        $browser->type('@title', $packageTitle)
            ->type('@instructions', 'Do this stuff and then that stuff');
    }
}

$browser->visit(new CreatePackage)
    ->fillBasicFields('Greatest Package Ever')
    ->press('Create Package')
    ->assertSee('Greatest Package Ever');

```

Компоненты. Если хотите использовать функциональность, предлагаемую страницами Dusk, без привязки к конкретному URL-адресу, то лучше применять *компоненты* Dusk. Внутренняя структура этих классов очень напоминает страницы, однако вместо привязки к URL-адресу каждый из них привязывается к селектору.

На сайте NovaPackages.com есть небольшой компонент Vue для присвоения пакетам рейтингов и их отображения. Сделаем для него соответствующий компонент Dusk:

```
php artisan dusk:component RatingWidget
```

Пример 12.36 демонстрирует сгенерированный компонент.

Пример 12.36. Стандартный код генерируемого компонента Dusk

```
<?php
```

```
namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class RatingWidget extends BaseComponent
{
    /**
     * Получаем корневой селектор для компонента
     *
     * @return string
     */
    public function selector()
    {
        return '#selector';
    }

    /**
     * Убеждаемся в том, что страница браузера содержит компонент
     *
     * @param Browser $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Получаем псевдонимы элементов для компонента
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@element' => '#selector',
        ];
    }
}
```

Здесь почти то же самое, что в случае страницы Dusk, с тем отличием, что весь код инкапсулируется не в URL-адрес, а в HTML-элемент. Все остальное выглядит практически аналогично. В примере 12.37 показано, как будет выглядеть наш виджет рейтингов в виде компонента Dusk.

Пример 12.37. Компонент Dusk для виджета рейтингов

```
class RatingWidget extends BaseComponent
{
    public function selector()
    {
        return '.rating-widget';
    }

    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    public function elements()
    {
        return [
            '@5-star' => '.five-star-rating',
            '@4-star' => '.four-star-rating',
            '@3-star' => '.three-star-rating',
            '@2-star' => '.two-star-rating',
            '@1-star' => '.one-star-rating',
            '@average' => '.average-rating',
            '@mine' => '.current-user-rating',
        ];
    }

    public function ratePackage(Browser $browser, $rating)
    {
        $browser->click("@{{$rating}-star}")
            ->assertSeeIn('@mine', $rating);
    }
}
```

Используются компоненты точно так же, как и страницы, в чем можно убедиться в примере 12.38.

Пример 12.38. Использование компонентов Dusk

```
$browser->visit('/packages/tightenco/nova-stock-picker')
->within(new RatingWidget, function ($browser) {
    $browser->ratePackage(2);
    $browser->assertSeeIn('@average', 2);
});
```

На этом мы закончим краткий обзор того, что можно сделать с помощью Dusk. В документации по Dusk (<http://bit.ly/2JF0POY>) вы найдете гораздо больше информации — утверждений, исключений из правил, подводных камней и примеров, поэтому рекомендуем ознакомиться с ней, если планируете работать с Dusk.

Резюме

Хотя Laravel позволяет использовать любой современный PHP-фреймворк для тестирования, он оптимизирован для использования PHPUnit (особенно если ваши тесты расширяют класс `TestCase` Laravel). С помощью среды тестирования приложений фреймворка можно легко отправлять поддельные HTTP-запросы и консольные команды вашему приложению и проверять результаты.

С Laravel легко использовать в тестах обширные возможности взаимодействия и проверки базы данных, кэша, сессии, файловой системы, электронной почты и многих других систем. Многие из них имеют встроенные методы для их подделки, что еще больше упрощает их тестирование. Вы можете протестировать DOM-модель и взаимодействие с браузером, применяя пакет `BrowserKit Testing` или `Dusk`.

На случай, если вам потребуются имитации, заглушки, шпионы, макеты или другие подобные инструменты, в состав Laravel включен пакет `Mockery`. Однако философия тестирования Laravel сводится к тому, чтобы по возможности использовать реальные взаимодействующие объекты. Имитируйте что-либо лишь в том случае, когда без этого нельзя обойтись.

13

Создание API

В число задач, которые наиболее часто приходится решать разработчикам приложений Laravel, входит создание API. Как правило, на базе JSON и REST или REST-подобного для обеспечения взаимодействия стороннего ПО с данными приложений Laravel.

Laravel многократно упрощает работу с форматом JSON. Структура его контроллеров ресурсов изначально ориентирована на использование команд и шаблонов REST. В этой главе вы узнаете о некоторых основных концепциях создания API, инструментах для создания API, предлагаемых фреймворком, а также о некоторых внешних инструментах и организационных системах, которые могут потребоваться при написании вашего первого API на базе Laravel.

Базовые сведения о REST-подобных API на базе JSON

REST (Representational State Transfer, передача состояния представления) — это архитектурный стиль для создания API. Строго говоря, REST может пониматься и как очень широкое понятие, применимое к Интернету во всей его совокупности, и как нечто узкоспецифичное до такой степени, что его фактически *никто* не использует. Поэтому не будем слишком заострять внимание на определении и дискутировать относительно его точности. В мире Laravel под RESTful и REST-подобными APIAPIAPI обычно подразумеваются API, обладающие следующим набором общих характеристик.

- ❑ Их структура ориентирована на использование «ресурсов», которые могут быть однозначно представлены с помощью URI-адресов, таких как `/cats` для всех кошек, `/cats/15` для одной кошки с номером 15 и т. д.
- ❑ Взаимодействие с ресурсами, как правило, осуществляется с помощью команд HTTP (`GET /cats/15` или `DELETE /cats/15`).

- ❑ Они не сохраняют состояние. Это означает, что при переходе от одного запроса к другому не сохраняется аутентификация сессии. Для каждого запроса нужна отдельная операция аутентификации.
- ❑ Они кэшируемые и последовательные. То есть все запросы (за исключением относящихся к аутентифицированному пользователю) должны возвращать одинаковый результат вне зависимости от источника запроса.
- ❑ Они возвращают данные в формате JSON.

Наиболее распространенный подход к созданию API — использовать уникальную структуру URL-адреса для каждой из моделей Eloquent, экспонируемых как ресурс API, и позволить пользователям взаимодействовать с этим ресурсом посредством конкретных команд, возвращающих данные в формате JSON. Пример 13.1 демонстрирует несколько вариантов применения этого подхода.

Пример 13.1. Типичная структура конечных точек API на базе REST

GET /api/cats

```
[
  {
    id: 1,
    name: 'Fluffy'
  },
  {
    id: 2,
    name: 'Killer'
  }
]
```

GET /api/cats/2

```
{
  id: 2,
  name: 'Killer'
}
```

DELETE /api/cats/2
(deletes cat)

POST /api/cats with body:

```
{
  name: 'Mr Bigglesworth'
}
(creates new cat)
```

PATCH /api/cats/3 with body:

```
{
  name: 'Mr. Bigglesworth'
}
(updates cat)
```

Вы уже знаете, какой набор взаимодействий в основном должны осуществлять наши API. Теперь посмотрим, как этого добиться с помощью Laravel.

Организация контроллеров и возвращаемые JSON-сообщения

Контроллеры ресурсов API в Laravel — это обычные контроллеры ресурсов (см. подраздел «Контроллеры ресурсов» на с. 71), модифицированные для использования маршрутов API, соответствующих стилю RESTful. Например, у них нет методов `create()` и `edit()`, неуместных в случае API. С этого и начнем. Сначала создадим новый контроллер для нашего ресурса с маршрутом `/api/dogs`:

```
php artisan make:controller Api\DogsController --api
```



Модификация символов обратной косой черты в командах Artisan до версии Laravel 5.3

В версиях Laravel до 5.3 в командах Artisan требовалось модифицировать символ `\`, используемый в качестве разделителя пространства имен, прямой косой чертой, как показано ниже:

```
php artisan make:controller Api/\DogsController -api
```

5.3 В проектах, использующих версии Laravel до 5.5, отсутствуют концепции контроллеров и маршрутов ресурсов API. Вместо них можно по-прежнему применять обычные контроллеры и маршруты ресурсов. Они отличаются лишь наличием нескольких относящихся к представлениям маршрутов, которые не используются в API. Пример 13.2 демонстрирует контроллер ресурсов API.

Пример 13.2. Сгенерированный контроллер ресурсов API

```
<?php
```

```
namespace App\Http\Controllers\Api;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class DogsController extends Controller
{
    /**
     * Отображение списка ресурсов
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }
}
```



```
/**
 * Сохранение вновь созданного ресурса в хранилище
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
}

/**
 * Отображение указанного ресурса
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}

/**
 * Обновление указанного ресурса в хранилище
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    //
}

/**
 * Удаление указанного ресурса из хранилища
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    //
}
}
```

Фактически все рассказали блоки документации. Метод `index()` отображает всех собак, `show()` — одну собаку, `store()` сохраняет новую собаку, `update()` обновляет и `destroy()` удаляет ее.

Быстро создадим модель и миграцию для работы:

```
php artisan make:model Dog --migration
php artisan migrate
```

Отлично! Теперь можно заполнить методы нашего контроллера.



Необходимые условия для работы кода этих примеров, касающиеся базы данных

Чтобы код приводимых здесь примеров действительно заработал, в миграцию нужно добавить метод `string()` для столбцов с именами `name` и `breed`, а также добавить эти столбцы в свойство `fillable` модели Eloquent или задать свойство `guarded` этой модели равным пустому массиву (`[]`).

Теперь можно воспользоваться одной из удобных возможностей Eloquent. При отображении коллекции результатов Eloquent она автоматически преобразуется в формат JSON (для этого используется магический метод `toString()`). То есть если при возвращении коллекции результатов от маршрута в действительности будут возвращаться данные в формате JSON. Таким образом, как показывает пример 13.3, необходимый код будет выглядеть предельно просто.

Пример 13.3. Пример контроллера ресурсов API для объекта Dog

```
...
class DogsController extends Controller
{
    public function index()
    {
        return Dog::all();
    }

    public function store(Request $request)
    {
        return Dog::create($request->only(['name', 'breed']));
    }

    public function show($id)
    {
        return Dog::findOrFail($id);
    }

    public function update(Request $request, $id)
    {
        $dog = Dog::findOrFail($id);
        $dog->update($request->only(['name', 'breed']));
        return $dog;
    }

    public function destroy($id)
    {
        Dog::findOrFail($id)->delete();
    }
}
```

Пример 13.4 показывает, как выполнить привязку этого контроллера в файле маршрутов. Мы можем автоматически отобразить все эти методы по умолча-

нию на соответствующие им маршруты и команды HTTP с помощью метода `Route::apiResource()`.

Пример 13.4. Привязка маршрутов к контроллеру ресурсов

```
// routes/api.php
Route::namespace('Api')->group(function () {
    Route::apiResource('dogs', 'DogsController');
});
```

Готово! Вы создали свой первый API на базе RESTful в Laravel. Конечно, еще нужно разобраться с множеством других нюансов, включая разбивку на страницы, сортировку, аутентификацию и более определенные заголовки ответов. Но основа для всего остального положена.

Чтение и отправка заголовков

API на базе REST часто производят чтение и отправку не относящейся к содержанию информации с помощью заголовков. Например, любой запрос к API сервиса GitHub возвращает заголовки с данными об ограничении частоты запросов для текущего пользователя:

```
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4987
X-RateLimit-Reset: 1350085394
```

ЗАГОЛОВКИ X-*

Может возникнуть вопрос, почему заголовки сервиса GitHub с данными об ограничении частоты запросов начинаются с символов X-. Особенно если вы сравните их с другими заголовками, возвращаемыми тем же запросом:

```
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 12 Oct 2012 23:33:14 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Status: 200 OK
ETag: "a00049ba79152d03380c34652f2cb612"
X-GitHub-Media-Type: github.v3
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4987
X-RateLimit-Reset: 1350085394
Content-Length: 5
Cache-Control: max-age=0, private, must-revalidate
X-Content-Type-Options: nosniff
```

Любой заголовок, название которого начинается с X-, не входит в спецификацию протокола HTTP. Это может быть абсолютно произвольный заголовок (например, `X-How-Much-Matt-Loves-This-Page`) либо какой-то общепринятый, еще не вошедший в спецификацию (скажем, `X-Requested-With`).

Аналогичным образом многие API позволяют разработчикам настраивать свои запросы с помощью заголовков. Например, применяя API сервиса GitHub, можно легко указывать, какая версия API нужна в вашем случае, используя заголовок **Accept**:

Accept: `application/vnd.github.v3+json`

Если вместо `v3` вы напишете `v2`, то сервис GitHub передаст ваш запрос версии 2 своего API.

Кратко расскажу, как производится чтение и отправка заголовков в Laravel.

Отправка заголовков ответа в Laravel

Мы немного касались этой темы в главе 10. Если у вас есть объект ответа, можно добавить заголовок, используя метод `header($headerName, $headerValue)`, как показано в примере 13.5.

Пример 13.5. Добавление заголовка ответа в Laravel

```
Route::get('dogs', function () {
    return response(Dog::all())
        ->header('X-Greatness-Index', 12);
});
```

Вот так, легко и просто!

Чтение заголовков запроса в Laravel

Если у вас есть входящий запрос, то прочитать нужный заголовок можно столь же легко. Это демонстрирует пример 13.6.

Пример 13.6. Чтение заголовка запроса в Laravel

```
Route::get('dogs', function (Request $request) {
    var_dump($request->header('Accept'));
});
```

Теперь вы умеете читать заголовки входящих запросов и определять заголовки ответов API. Поговорим о том, какие настройки API еще могут потребоваться.

Разбивка на страницы в Eloquent

Пагинация — это одна из тех функций, для которой нужно в первую очередь подумать о снабжении API специальными инструкциями. Библиотека Eloquent «из коробки» предлагает систему разбивки на страницы, которая подключается непосредственно к параметрам любого запроса страницы. Мы уже немного говорили об этом в главе 6, но будет нелишним вспомнить еще раз.

Любой вызов Eloquent предоставляет метод `paginate()`, которому в качестве параметра можно передать желаемое количество элементов, отображаемых на одной

странице. При этом Eloquent проверит, определен ли в URL-адресе параметр `page`. Если да, интерпретирует его как указатель того, на сколько страниц пользователь продвинулся в постраничном списке.

Чтобы подготовить маршрут API к автоматической разбивке на страницы, определите запросы Eloquent для этого маршрута с помощью метода `paginate()`, а не `all()` или `get()` — примерно так, как показано в примере 13.7.

Пример 13.7. Постраничный маршрут API

```
Route::get('dogs', function () {  
    return Dog::paginate(20);  
});
```

Мы указали, что Eloquent будет извлекать из базы данных по 20 результатов. *Какие именно 20 результатов* это должны быть, фреймворку Laravel будет указывать значение параметра запроса `page`:

```
GET /dogs          - Return results 1-20  
GET /dogs?page=1   - Return results 1-20  
GET /dogs?page=2   - Return results 21-40
```

Обратите внимание, что метод `paginate()` можно использовать в вызовах генератора запросов, как показано в примере 13.8.

Пример 13.8. Использование метода `paginate()` в вызове генератора запросов

```
Route::get('dogs', function () {  
    return DB::table('dogs')->paginate(20);  
});
```

Данный вызов не обеспечит простое возвращение 20 результатов при его преобразовании в формат JSON. Вместо этого он создаст объект ответа, который *наряду* с постраничными данными автоматически передаст конечному пользователю полезные сведения о пагинации. Как может выглядеть возвращаемый данным вызовом ответ, показано в примере 13.9. Для экономии места я ограничился тремя записями.

Пример 13.9. Пример данных, возвращаемых постраничным вызовом базы данных

```
{  
    "current_page": 1,  
    "data": [  
        {  
            "name": "Fido"  
        },  
        {  
            "name": "Pickles"  
        },  
        {  
            "name": "Spot"  
        }  
    ],  
    "first_page_url": "http://myapp.com/api/dogs?page=1",  
    "last_page_url": null,  
    "next_page_url": null,  
    "previous_page_url": null,  
    "total": 3  
}
```

```
"from": 1,  
"last_page": 2,  
"last_page_url": "http://myapp.com/api/dogs?page=2",  
"next_page_url": "http://myapp.com/api/dogs?page=2",  
"path": "http://myapp.com/api/dogs",  
"per_page": 2,  
"prev_page_url": null,  
"to": 2,  
"total": 4  
}
```

Сортировка и фильтрация

Если для разбивки на страницы в Laravel предусмотрены некоторые соглашения и встроенные инструменты, то для сортировки не предлагается никаких инструментов и вам придется решать эту задачу самостоятельно. Я приведу здесь небольшой пример кода, в котором параметры запроса определены в стиле, предписываемом спецификацией JSON API (см. ее описание в приведенной ниже врезке).

СПЕЦИФИКАЦИЯ JSON API

Спецификация JSON API (<http://jsonapi.org/>) является стандартом для выполнения многих распространенных задач при создании API на базе формата JSON, таких как фильтрация, сортировка, разбивка на страницы, аутентификация, встраивание кода, использование ссылок и метаданных и т. д.

Хотя пагинация, предлагаемая Laravel по умолчанию, *абсолютно точно* не соответствует спецификации JSON API, но это шаг в правильном направлении. Остальные предписания спецификации JSON API относятся к категории вещей, которые (при наличии желания) требуется реализовать вручную.

В качестве примера ниже приведен фрагмент спецификации JSON API, который услужливо подсказывает, какой должна быть структура документа при возвращении ошибок.

Документ **ДОЛЖЕН** содержать хотя бы один из следующих членов верхнего уровня:

- **data** — «первичные данные» документа;
- **errors** — массив объектов ошибок;
- **meta** — метаобъект с нестандартной метаинформацией.

Члены **data** и **errors** **НЕ ДОЛЖНЫ** сосуществовать в одном документе.

Какие бы преимущества не несло следование спецификации JSON API, я должен предупредить, что для ее эффективного использования нужно провести основательную подготовку. В примерах я не следую данной спецификации полностью, но руковожусь ее основными идеями.

Сортировка результатов API

Для начала определим возможность сортировки наших результатов. В примере 13.10 мы начнем с сортировки только по одному столбцу и направлению.

Пример 13.10. Простейшая сортировка API

```
// Обрабатываем /dogs?sort=name
Route::get('dogs', function (Request $request) {
    // Получаем параметр сортировки (или откатываемся
    // к значению по умолчанию "name")
    $sortColumn = $request->input('sort', 'name');
    return Dog::orderBy($sortColumn)->paginate(20);
});
```

В примере 13.11 к этому добавляется возможность инвертировать направление сортировки (с помощью выражения вида `?sort=-weight`).

Пример 13.11. Сортировка API по одному столбцу с контролем над направлением сортировки

```
// Обрабатываем /dogs?sort=name и /dogs?sort=-name
Route::get('dogs', function (Request $request) {
    // Получаем параметр сортировки (или откатываемся
    // к значению по умолчанию "name")
    $sortColumn = $request->input('sort', 'name');

    // Устанавливаем направление сортировки в зависимости от наличия
    // перед значением ключа знака -,
    // используя хелпер starts_with() Laravel
    $sortDirection = starts_with($sortColumn, '-') ? 'desc' : 'asc';
    $sortColumn = ltrim($sortColumn, '-');

    return Dog::orderBy($sortColumn, $sortDirection)
        ->paginate(20);
});
```

В примере 13.12 это реализуется для нескольких столбцов (с использованием выражения вида `?sort=name,-weight`).

Пример 13.12. Сортировка в стиле, предписываемом спецификацией JSON API

```
// Обрабатываем ?sort=name,-weight
Route::get('dogs', function (Request $request) {
    // Получаем параметр запроса и преобразуем его в массив,
    // разделенный запятыми
    $sorts = explode(',', $request->input('sort', ''));

    // Создаем запрос
    $query = Dog::query();

    // Поочередно добавляем элементы массива sorts
    foreach ($sorts as $sortColumn) {
```

```
$sortDirection = starts_with($sortColumn, '-') ? 'desc' : 'asc';
$sortColumn = ltrim($sortColumn, '-');

$query->orderBy($sortColumn, $sortDirection);
}

// Возвращаем результаты
return $query->paginate(20);
});
```

Это не самая простая процедура. Некоторые повторяющиеся части потребуются выделить в ряд вспомогательных инструментов. При этом мы последовательно расширяем возможности настройки API, используя логически обоснованные и простые функции.

Фильтрация результатов API

Еще одна распространенная задача при создании API — фильтрация с получением на выходе только определенного подмножества данных. Например, клиент может запросить список собак породы чихуахуа.

Спецификация JSON API не дает здесь каких-либо важных указаний в отношении синтаксиса, указывая только, что мы должны использовать параметр запроса **filter**. Возьмем за образец синтаксис сортировки и разместим все в одном ключе выражением вида **?filter=breed:chihuahua**. Итог показан в примере 13.13.

Пример 13.13. Одиночный фильтр результатов API

```
Route::get('dogs', function () {
    $query = Dog::query();

    $query->when(request()->filled('filter'), function ($query) {
        [$criteria, $value] = explode(':', request('filter'));
        return $query->where($criteria, $value);
    });

    return $query->paginate(20);
});
```

В примере 13.13 мы используем хелпер `request()`, а не внедряем экземпляр `$request`. Хотя оба способа одинаковы, иногда для замыканий лучше подходит хелпер `request()`, поскольку при этом не нужно передавать переменные вручную.



Условная модификация запроса до Laravel 5.2

В проектах, использующих версии Laravel до 5.2 потребуется заменить вызов метода `$query->when()` обычным оператором `if` языка PHP.

В примере 13.14 реализуем множественную фильтрацию с помощью выражения вида `?filter=breed:chihuahua,color:brown`.

Пример 13.14. Множественная фильтрация результатов API

```
Route::get('dogs', function (Request $request) {
    $query = Dog::query();

    $query->when(request()->filled('filter'), function ($query) {
        $filters = explode(',', request('filter'));

        foreach ($filters as $filter) {
            [$criteria, $value] = explode(':', $filter);
            $query->where($criteria, $value);
        }

        return $query;
    });

    return $query->paginate(20);
});
```

Преобразование результатов

Мы рассмотрели, как можно сортировать и фильтровать наши наборы результатов. Однако при этом мы пока используем механизм JSON-сериализации библиотеки Eloquent, что означает возвращение всех полей в любой модели.

Библиотека Eloquent предлагает вспомогательные инструменты для определения того, какие поля следует представить при сериализации массива. Об этом подробно написано в главе 5, но основная суть в том, что если вы определите в своем классе Eloquent содержащее массив свойство `$hidden`, то любое поле, указанное там, не будет представлено в сериализованном выводе модели. Альтернативный способ — определить массив `$visible`, указывающий, какие поля следует представить в результатах. Можно переписать или имитировать функцию `toArray()` в своей модели, определив собственный формат вывода.

Еще один распространенный подход — создать преобразователи для каждого типа данных. Удобство в большей степени контроля, отделении относящейся к API логики от модели и возможности предоставить более последовательный API даже в случае изменения моделей и их связей.

Для этого предусмотрен прекрасно справляющийся со своей задачей, но достаточно сложный пакет Fractal (<http://bit.ly/2fEt8Nr>), который определяет ряд вспомогательных структур и классов для преобразования данных.

5.5

В Laravel 5.5 введена новая концепция ресурсов API, которая удовлетворяет потребности большинства API в плане преобразования и сбора результатов.

Поэтому, если вы используете Laravel 5.5 или более новую версию, можете сразу перейти к разделу «Ресурсы API» на с. 365. Если же Laravel 5.4 или более ранняя версия, продолжите последовательное чтение.

Создание собственного преобразователя

Основная идея применения преобразователя в том, что каждый экземпляр нашей модели должен запускаться с использованием еще одного класса, который будет *преобразовывать* свои данные к другому виду. Этот класс может добавлять, переименовывать и удалять поля, манипулировать ими, добавлять вложенные дочерние элементы и т. д. Начнем с простого примера (пример 13.15).

Пример 13.15. Простой преобразователь

```
Route::get('users/{id}', function ($userId) {
    return (new UserTransformer(User::findOrFail($userId)));
});
```

```
class UserTransformer
{
    protected $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function toArray()
    {
        return [
            'id' => $this->user->id,
            'name' => sprintf(
                "%s %s",
                $this->user->first_name,
                $this->user->last_name
            ),
            'friendsCount' => $this->user->friends->count(),
        ];
    }

    public function toJson()
    {
        return json_encode($this->toArray());
    }

    public function __toString()
    {
        return $this->toJson();
    }
}
```



Традиционные преобразователи

Если бы мы решили использовать более традиционный преобразователь, то его нужно было бы снабдить методом `transform()`, который принимал бы параметр `$user` и выдавал массив или непосредственно данные в формате JSON.

Однако я уже в течение нескольких лет использую данный шаблон, который мы иногда называем «объекты API», и успел убедиться, что он гораздо мощнее и гибче традиционного подхода.

Как вы можете видеть из примера 13.15, преобразователи принимают изменяемую ими модель в качестве параметра, после чего производят манипуляции над ней и ее связями, формируя окончательные выходные данные, которые нужно отправить API.

Вложение связей пользовательских преобразователей

Вопрос о том, как следует производить вложение связей в API и стоит ли это делать, является предметом больших дискуссий. Опытные люди уже достаточно много написали на эту тему. Я рекомендую прочитать книгу Фила Стерджена *Build APIs You Won't Hate* (Leanpub) по адресу <https://apisyouwonthate.com/>. В ней подробно освещена данная тема и в целом создание API на базе REST.

Существует несколько основных подходов к вложению связей. В примерах ниже предполагается, что у вас есть основной ресурс `user` и связанный с ним ресурс `friend`.

- ☐ Вкладывать связанные ресурсы непосредственно в основной ресурс (например, в ресурс `users/5` вкладываются его друзья).
- ☐ Вкладывать в основной ресурс только внешние ключи (скажем, в ресурс `users/5` вкладывается массив идентификаторов друзей).
- ☐ Предоставить пользователю возможность запрашивать связанные ресурсы с фильтрацией по основному ресурсу (к примеру, выражение `/friends?user=5` предписывает выдать всех друзей, связанных с пользователем 5).
- ☐ Создать подресурс (предположим, `/users/5/friends`).
- ☐ Предоставить возможность опционального вложения (например, вложение не производится в выражении `/users/5`, но производится в выражениях `/users/5?include=friends` и `/users/5?include=friends,dogs`).

Предположим, нужно производить опциональное вложение связанных элементов. Как мы можем это реализовать? Пример 13.15 с преобразователем дает определенную фору. Отредактируем этот код, как показано в примере 13.16, чтобы добавить возможность опционального вложения подресурса.

Пример 13.16. Реализация возможности опционального вложения подресурса в преобразователь

```
// Поддержка выражений вида myapp.com/api/users/15?include=friends,bookmarks
Route::get('users/{id}', function ($userId, Request $request) {
    // Получаем параметр запроса include и разделяем его запятыми
    $includes = explode(',', $request->input('include', ''));
    // Передаем пользователя и вложения преобразователю пользователей
    return (new UserTransformer(User::findOrFail($userId), $includes));
});

class UserTransformer
{
    protected $user;
    protected $includes;

    public function __construct($user, $includes = [])
    {
        $this->user = $user;
        $this->includes = $includes;
    }

    public function toArray()
    {
        $append = [];

        if (in_array('friends', $this->includes)) {
            // Если у вас несколько вложений, нужно выполнить генерализацию
            $append['friends'] = $this->user->friends->map(function ($friend) {
                return (new FriendTransformer($friend))->toArray();
            });
        }

        return array_merge([
            'id' => $this->user->id,
            'name' => sprintf(
                "%s %s",
                $this->user->first_name,
                $this->user->last_name
            ),
        ], $append);
    }
}
...
```

Мы подробно рассмотрим функциональность метода `map()` при обсуждении коллекций в главе 17, а все остальное в этом коде должно быть вам знакомо.

В маршруте мы разделяем параметр запроса `include` запятыми и передаем его нашему преобразователю. Пока он может работать только с вложением `friends`, но, прибегнув к абстракции, можно обеспечить поддержку и других вложений. Когда пользователь запрашивает `friends`, преобразователь сопоставляет все объ-

екты `friend` (используя связь `hasMany` с `friend` в модели `user`), передает их методу `FriendTransformer()` и включает массив из всех преобразованных объектов `friends` в ответ.

Ресурсы API

5.5 Если вы используете версии до Laravel 5.5, можно пропустить этот раздел и сразу перейти к разделу «Аутентификация API с помощью Laravel Passport» на с. 370.

В прошлом одной из проблем, с которой мы сталкивались при разработке API в Laravel, был способ преобразования возвращаемых данных. В простейшем случае API могли просто возвращать объекты Eloquent в формате JSON, но потребности большинства API быстро выходят за рамки этой структуры. Так как же выполнять преобразование выходных объектов Eloquent в нужный формат? Что, если потребуется внедрять другие ресурсы (возможно, опционально), или добавлять вычисляемое поле, или скрывать некоторые поля для API, оставив их видимыми в остальном JSON-выводе? Ответом является преобразователь для API.

В Laravel 5.5 и более новых версиях в нашем распоряжении имеются так называемые Eloquent-ресурсы API, которые представляют собой структуры, определяющие способ преобразования объекта Eloquent (или коллекции объектов Eloquent), относящегося к определенному классу, в результаты API. Например, ваша модель Eloquent `Dog` теперь имеет ресурс `Dog`, который обеспечивает преобразование каждого экземпляра класса `Dog` в соответствующий объект ответа API с данными о собаке.

Создание класса ресурса

Пошагово разберем данный пример с классом `Dog`, чтобы посмотреть, как осуществляется преобразование вывода API. Первое, что вы должны сделать, — это создать свой первый ресурс с помощью команды `Artisan make:resource`:

```
php artisan make:resource Dog
```

Эта команда создаст новый класс в файле `app/Http/Resources/Dog.php` с единственным методом `toArray()` (пример 13.17).

Пример 13.17. Сгенерированный ресурс API

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class Dog extends JsonResource
{
```

```

/**
 * Преобразование ресурса в массив
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return parent::toArray($request);
}
}

```

Метод `toArray()` имеет доступ к двум важным фрагментам данных. Во-первых, он может обращаться к объекту `Illuminate Request`, что позволяет настроить ответ, исходя из параметров запроса, заголовков и других важных данных. Во-вторых, он может обращаться ко всему преобразуемому объекту `Eloquent` путем вызова его свойств и методов в переменной `$this`, как показано в примере 13.18.

Пример 13.18. Простой ресурс API для модели Dog

```

class Dog extends JsonResource
{
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'breed' => $this->breed,
        ];
    }
}

```

Чтобы использовать этот новый ресурс, нужно обновить любую конечную точку API, которая возвращает один объект `Dog`, обернув ответ в новый ресурс, как показано в примере 13.19.

Пример 13.19. Использование простого ресурса Dog

```

use App\Dog;
use App\Http\Resources\Dog as DogResource;

Route::get('dogs/{dogId}', function ($dogId) {
    return new DogResource(Dog::find($dogId));
});

```

Коллекции ресурсов

Теперь поговорим, что делать, когда конечная точка API должна возвращать несколько сущностей. Этого можно добиться с помощью метода `collection()` ресурса API, как показано в примере 13.20.

Пример 13.20. Использование предлагаемого по умолчанию метода `collection()` ресурса API

```
use App\Dog;
use App\Http\Resources\Dog as DogResource;
Route::get('dogs', function () {
    return DogResource::collection(Dog::all());
});
```

Данный метод последовательно перебирает все переданные ему элементы, преобразуя каждый из них в ресурс `DogResource` API, после чего возвращает коллекцию.

Для многих API этого достаточно. Однако если нужно каким-либо образом модифицировать структуру или дополнить коллекцию ответов метаданными, вместо этого следует создать пользовательскую коллекцию ресурсов API.

Потребуется еще раз воспользоваться командой `Artisan make:resource`. На этот раз мы передадим ей имя `DogCollection`, что послужит для `Laravel` сигналом, что это не просто ресурс API, а коллекция ресурсов API:

```
php artisan make:resource DogCollection
```

Эта команда сгенерирует новый файл `app/Http/Resources/DogCollection.php`, который очень похож на файл API-ресурса и будет снова содержать единственный метод `toArray()` (пример 13.21).

Пример 13.21. Сгенерированная коллекция ресурсов API

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class DogCollection extends ResourceCollection
{
    /**
     * Преобразование коллекции ресурсов в массив
     *
     * @param \Illuminate\Http\Request $request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }
}
```

Как и в случае API-ресурса, у нас есть доступ к запросу и базовым данным. Но, в отличие от API-ресурса, здесь не один объект, а коллекция. Соответственно, будем обращаться к этой (уже преобразованной) коллекции с помощью выражения `$this->collection`. Как это выглядит, показано в примере 13.22.

Пример 13.22. Простая коллекция ресурсов API для модели Dog

```
class DogCollection extends ResourceCollection
{
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => route('dogs.index'),
            ],
        ];
    }
}
```

Вложение связей

Один из самых сложных аспектов любого API — механизм вложения связей. В случае ресурсов API проще всего добавить в возвращаемый массив соответствующий ключ и присвоить ему коллекцию ресурсов API, как показано в примере 13.23.

Пример 13.23. Простой пример вложения связей API

```
public function toArray()
{
    return [
        'name' => $this->name,
        'breed' => $this->breed,
        'friends' => DogResource::collection($this->friends),
    ];
}
```

Иногда нужно сделать это свойство условным — например, его можно вкладывать, только если на него сделан запрос или свойство загружено путем активной загрузки в переданном объекте Eloquent. Взгляните на пример 13.24.

Пример 13.24. Условная загрузка связей API

```
public function toArray()
{
    return [
        'name' => $this->name,
        'breed' => $this->breed,
        // Загружаем эту связь, только если она уже задана
        // путем активной загрузки
        'bones' => BoneResource::collection($this->whenLoaded('bones')),
        // Или загружаем эту связь, только если она запрашивается в URL-адресе
        'bones' => $this->when(
            $request->get('include') == 'bones',
            BoneResource::collection($this->bones)
        ),
    ];
}
```


Применение разбивки на страницы к ресурсам API

Экземпляр разбивщика страниц можно передать так же, как ресурс коллекции моделей Eloquent. Взгляните на пример 13.25.

Пример 13.25. Передача экземпляра разбивщика страниц коллекции ресурсов API

```
Route::get('dogs', function () {  
    return new DogCollection(Dog::paginate(20));  
});
```

Если мы передадим экземпляр разбивщика страниц, то у преобразованного результата будут дополнительные ссылки, содержащие информацию постраничной разбивки (**first** (первая страница), **last** (последняя страница), **prev** (предыдущая страница) и **next** (следующая страница)) и метаданные обо всей коллекции.

Как выглядит такая информация, показано в примере 13.26. С помощью вызова `Dog::paginate(2)` я задал количество элементов на страницу равным 2 для наглядности работы ссылок.

Пример 13.26. Пример постраничного ответа ресурса со ссылками на страницы

```
{  
    "data": [  
        {  
            "name": "Pickles",  
            "breed": "Chorkie",  
        },  
        {  
            "name": "Gandalf",  
            "breed": "Golden Retriever Mix",  
        }  
    ],  
    "links": {  
        "first": "http://gooddogbrant.com/api/dogs?page=1",  
        "last": "http://gooddogbrant.com/api/dogs?page=3",  
        "prev": null,  
        "next": "http://gooddogbrant.com/api/dogs?page=2"  
    },  
    "meta": {  
        "current_page": 1,  
        "from": 1,  
        "last_page": 3,  
        "path": "http://gooddogbrant.com/api/dogs",  
        "per_page": 2,  
        "to": 2,  
        "total": 5  
    }  
}
```

Условное применение атрибутов

Можно указать, что определенные атрибуты должны применяться в вашем ответе только при выполнении определенного условия, как показано в примере 13.27.

Пример 13.27. Условное применение атрибутов

```
public function toArray($request)
{
    return [
        'name' => $this->name,
        'breed' => $this->breed,
        'rating' => $this->when(Auth::user()->canSeeRatings(), 12),
    ];
}
```

Другие настройки для ресурсов API

Иногда не совсем подходит предлагаемый по умолчанию способ обертывания свойства `data` или нужно добавить/модифицировать метаданные для ответов. Чтобы в подробностях узнать о настройке каждого аспекта ответов API, ознакомьтесь с документацией по адресу <http://bit.ly/2HP8xTU>.

Аутентификация API с помощью Laravel Passport

5.3 Большинству API требуется определенная форма аутентификации для доступа к некоторым или всем данным. В Laravel 5.2 появилась простая схема аутентификации с помощью токена, которую мы обсудим чуть позже. В Laravel 5.3 и более новых версиях мы получили новый инструмент с названием Passport (отдельный пакет, подгружаемый с помощью утилиты Composer), который позволяет легко установить в своем приложении полнофункциональный сервер OAuth 2.0 вместе с API и компонентами пользовательского интерфейса для управления клиентами и токенами.

Вводная информация о OAuth 2.0

Протокол OAuth является безоговорочным лидером по использованию в качестве системы аутентификации в API на базе RESTful. К сожалению, тема слишком сложна, чтобы ее можно было всесторонне рассмотреть в этой книге. Для подробного изучения рекомендую прочитать книгу Мэтта Фроста *Integrating Web Services with OAuth and PHP (php[architect])* по OAuth и PHP.

В основе протокола OAuth лежит следующая простейшая идея: поскольку API не сохраняют состояние, мы не можем использовать такой же способ аутенти-

фикации на основе сессии, как в обычных браузерных сессиях просмотра, когда пользователь входит в систему и его аутентифицированное состояние сохраняется в сессии для последующих просмотров. Вместо этого клиент API должен сделать однократный вызов конечной точки аутентификации и идентифицировать себя с помощью определенной процедуры опознавания. После ему возвращается токен, который он должен отправлять вместе с каждым последующим запросом (обычно через заголовок **Authorization**) для подтверждения идентичности.

Существует несколько разновидностей выдаваемого протоколом OAuth допуска. Это значит, что такое аутентификационное опознавание может определяться рядом различных сценариев и типов взаимодействия. Для различных проектов и типов конечных потребителей типы допуска могут различаться.

Если вы работаете с Laravel 5.1 или 5.2, подходит пакет с названием OAuth 2.0 Server for Laravel («Сервер OAuth 2.0 для Laravel») (<http://bit.ly/2e2lFYi>), который позволяет сравнительно легко добавить простейший сервер аутентификации OAuth 2.0 в приложение Laravel. Это вспомогательная прослойка между Laravel и PHP-пакетом с названием PHP OAuth 2.0 Server (<http://bit.ly/2f1dUyP>).

5.3 Если вы используете Laravel 5.3 или более новую версию, то все, что предлагает этот пакет вместе с другими возможностями, найдете в Passport, где более простые и мощные API- и пользовательский интерфейсы.

Установка пакета Passport

Поскольку Passport представляет собой отдельный пакет, сначала нужно его установить. Я конспективно изложу нужные действия, но более подробные указания по установке вы найдете в документации по адресу <http://bit.ly/2fEBjtk>.

Подгрузите этот пакет с помощью утилиты Composer:

```
composer require laravel/passport
```

Если вы используете версию до Laravel 5.5, добавьте класс `Laravel\Passport\PassportServiceProvider::class` в массив `providers` в файле `config/app.php`.

Passport импортирует ряд миграций, поэтому запустите их с помощью команды `php artisan migrate`, чтобы создать необходимые таблицы для клиентов, областей видимости и токенов OAuth.

Затем запустите установщик командой `php artisan passport:install`. Это обеспечит создание ключей шифрования для сервера OAuth (`storage/oauth-private.key` и `storage/oauth-public.key`) и внесение клиентов OAuth в базу данных для токенов личного доступа и допуска по паролю (которые мы рассмотрим чуть позже).

Далее импортируйте трейт `Laravel\Passport\HasApiTokens` в свою модель `User`. Тем самым вы добавите в каждый объект `User` отношения, связанные с клиентами

и токенами OAuth, а также ряд вспомогательных методов для работы с токенами. Затем напишите вызов метода `Laravel\Passport\Passport::routes()` в метод `boot()` провайдера `AuthServiceProvider`. Тем самым вы добавите следующие маршруты:

- ☐ `oauth/authorize;`
- ☐ `oauth/clients;`
- ☐ `oauth/clients/client_id;`
- ☐ `oauth/personal-access-tokens;`
- ☐ `oauth/personal-access-tokens/token_id;`
- ☐ `oauth/scopes;`
- ☐ `oauth/token;`
- ☐ `oauth/token/refresh;`
- ☐ `oauth/tokens;`
- ☐ `oauth/tokens/token_id.`

Наконец, найдите код гарда `api` в файле `config/auth.php`. По умолчанию этот гард использует драйвер `token` (который мы обсудим чуть позже). Укажите вместо него драйвер `passport`.

Вот вы и получили полнофункциональный сервер OAuth 2.0! Теперь можно создавать новые клиенты с помощью команды `php artisan passport:client` и использовать API для управления клиентами и токенами, доступный по префиксу маршрута `/oauth`.

Чтобы защитить маршрут системой аутентификации Passport, добавьте middleware `auth:api` в маршрут или группу маршрутов, как показано в примере 13.28.

Пример 13.28. Защита маршрута API с помощью middleware аутентификации Passport

```
// routes/api.php
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:api');
```

Чтобы аутентифицироваться для доступа к таким защищенным маршрутам, клиентскому приложению нужно передавать некоторый токен (мы вскоре рассмотрим его получение) в качестве токена `Bearer` в заголовке `Authorization`. Пример 13.29 показывает, как это может выглядеть в случае выполнения запроса с помощью библиотеки `Guzzle HTTP`.

Пример 13.29. Пример выполнения запроса к API с использованием токена Bearer

```
$http = new GuzzleHttp\Client;
$response = $http->request('GET', 'http://tweeter.test/api/user', [
    'headers' => [
```

```
        'Accept' => 'application/json',  
        'Authorization' => 'Bearer ' . $accessToken,  
    ],  
});
```

Теперь подробнее остановимся на том, как все это работает.

API пакета Passport

Passport экспонирует API в приложении по префиксу маршрута `/oauth`. Этот интерфейс служит для двух основных функций: авторизации пользователей посредством схем авторизации OAuth 2.0 (`/oauth/authorize` и `/oauth/token`) и предоставления пользователям возможности управлять своими клиентами и токенами (остальными маршрутами).

Обратите внимание на это важное отличие, особенно если незнакомы с OAuth. Каждый сервер OAuth должен экспонировать возможность прохождения потребителями аутентификации на сервере, и это все его назначение. Однако Passport *также* экспонирует API для управления состоянием клиентов и токенов вашего сервера OAuth. Это означает, что вы можете легко создать клиентскую часть, позволяющую пользователям управлять своей информацией в вашем приложении OAuth. Вместе с Passport уже предоставляются такие управляющие компоненты на базе Vue, которые вы можете использовать в неизменном или модифицированном виде.

Далее мы подробно поговорим о маршрутах данного API для управления клиентами и токенами, а также поставляемых вместе с Passport компонентах на базе Vue. Но сначала остановимся на способах аутентификации, которые вы можете использовать при предоставлении пользователям доступа к своему API, защищенному с помощью Passport.

Типы допуска, предлагаемые пакетом Passport

Passport предлагает четыре способа аутентификации пользователей. Два из них — традиционные типы допуска OAuth 2.0 (допуск по паролю и по коду авторизации), а два других — вспомогательные методы из числа уникальных особенностей Passport (использование личного токена и токена синхронизатора).

Допуск по паролю

Допуск по паролю используется не столь часто, как по коду авторизации, но он гораздо проще последнего. Если нужно, чтобы пользователи могли проходить аутентификацию для доступа к вашему API путем непосредственного ввода имени пользователя и пароля (например, если у вас есть корпоративное мобильное приложение с собственным API), то подходит допуск по паролю.



Создание клиента с допуском по паролю

Чтобы можно было использовать схему допуска по паролю, в вашей базе данных должен присутствовать клиент с допуском по паролю. Потому что каждый запрос к серверу OAuth должен выполняться клиентом. Обычно клиент идентифицирует то приложение или сайт, для доступа к которому аутентифицируется пользователь. Например, в случае использования сети Facebook для аутентификации на стороннем сайте этот сайт будет клиентом.

Однако при использовании схемы допуска по паролю клиент не предоставляется вместе с запросом. Потому нужно создавать таковой, и это должен быть клиент с допуском по паролю. Он будет добавляться при выполнении команды `php artisan passport:install`, но, если по какой-либо причине нужно сгенерировать новый клиент с допуском по паролю, это можно сделать следующим образом:

```
$php artisan passport:client --password
```

```
What should we name the password grant client?
```

```
[My Application Password Grant Client]:
```

```
> Client_name
```

```
Password grant client created successfully.
```

```
Client ID: 3
```

```
Client Secret: Pg1EEzt18JAnFoUIM9n38Nqewg1aekB4rvFk2Pma
```

В случае допуска по паролю для получения токена нужно лишь одно действие: отправить учетные данные пользователя по маршруту `/oauth/token`, как показано в примере 13.30.

Пример 13.30. Создание запроса с допуском по паролю

```
// Файл routes/web.php в *приложении-потребителе*
Route::get('tweeter/password-grant-auth', function () {
    $http = new GuzzleHttp\Client;

    // Вызов сервера Tweeter – нашего OAuth-сервера на базе Passport
    $response = $http->post('http://tweeter.test/oauth/token', [
        'form_params' => [
            'grant_type' => 'password',
            'client_id' => config('tweeter.id'),
            'client_secret' => config('tweeter.secret'),
            'username' => 'matt@mattstauffer.co',
            'password' => 'my-tweeter-password',
            'scope' => '',
        ],
    ]);

    $thisUsersTokens = json_decode((string) $response->getBody(), true);
    // Выполнение определенных действий над токенами
});
```

Данный маршрут будет возвращать токены `access_token` (токен доступа) и `refresh_token` (токен обновления). После этого их можно сохранить, чтобы впоследствии

проходить аутентификацию для доступа к API (токен доступа) и запрашивать дополнительные токены (токен обновления).

Обратите внимание, что идентификатор и пароль для допуска по паролю будут размещены в таблице базы данных `oauth_clients` нашего приложения Passport, в строке с именем, совпадающим с названием нашего клиента допуска Passport. В этой таблице также будут присутствовать записи для двух клиентов, генерируемых по умолчанию при выполнении команды `passport:install`: Laravel Personal Access Client («клиент Laravel с личным доступом») и Laravel Password Grant Client («клиент Laravel с допуском по паролю»).

Допуск по коду авторизации

Эта наиболее часто используемая схема аутентификации OAuth 2.0 в то же время и самая сложная из поддерживаемых пакетом Passport. Предположим, нужно разработать приложение, представляющее собой Twitter для аудиозаписей. Назовем его Tweeter. Допустим, некто работает над другим сайтом, который представляет собой социальную сеть для любителей научной фантастики и называется SpaceBook. Разработчику сети SpaceBook нужно предоставить пользователям возможность встраивать свои данные из сети Tweeter в ленты новостей в SpaceBook. В таком случае нужно установить пакет Passport в нашем приложении Tweeter, чтобы другие приложения — как SpaceBook — могли предоставить своим пользователям возможность аутентифицироваться с помощью данных из Tweeter.

При использовании допуска по коду авторизации каждый сайт-потребитель — в данном случае SpaceBook — должен создавать клиент в нашем приложении с активизированным пакетом Passport. В большинстве случаев администраторы других сайтов будут иметь пользовательские учетные записи в сети Tweeter, и мы встроим туда инструменты, позволяющие им создавать клиенты. Но для начала можно вручную создать клиент для администраторов SpaceBook:

```
$php artisan passport:client
```

```
Which user ID should the client be assigned to?:
```

```
> 1 ❶
```

```
What should we name the client?:
```

```
> SpaceBook
```

```
Where should we redirect the request after authorization?
```

```
[http://tweeter.test/auth/callback]:
```

```
> http://spacebook.test/tweeter/callback
```

```
New client created successfully.
```

```
Client ID: 4
```

```
Client secret: 5rzqKpeCjIgz3MXpi3tjQ37HbNLLykrGwgmc18uH
```

❶ Каждый клиент должен быть назначен конкретному пользователю вашего приложения. Допустим, пользователь 1 — это разработчик сети SpaceBook. Он будет «владельцем» создаваемого нами клиента.

Таким образом, мы имеем идентификатор и пароль для клиента SpaceBook. В этой точке SpaceBook может использовать идентификатор и пароль для создания инструментов, позволяющих отдельному пользователю данной сети (одновременно являющемуся пользователем Tweeter) получить от Tweeter токен аутентификации и применять его в тех случаях, когда SpaceBook требуется вызывать API Tweeter от имени этого пользователя. Как это можно сделать, показано в примере 13.31. (В этом и следующих примерах предполагается, что SpaceBook тоже является приложением Laravel и его разработчик создал файл `config/tweeter.php`, который возвращает только что созданные нами идентификатор и пароль.)

Пример 13.31. Приложение-потребитель перенаправляет пользователя к нашему серверу OAuth

```
// В файле routes/web.php приложения SpaceBook:
Route::get('tweeter/redirect', function () {
    $query = http_build_query([
        'client_id' => config('tweeter.id'),
        'redirect_uri' => url('tweeter/callback'),
        'response_type' => 'code',
        'scope' => '',
    ]);

    // Создает строку следующего вида:
    // client_id={client_id}&redirect_uri={redirect_uri}&response_type=code

    return redirect('http://tweeter.test/oauth/authorize?' . $query);
});
```

Теперь при выборе пользователями этого маршрута в приложении SpaceBook они будут перенаправляться по маршруту пакета Passport `/oauth/authorize` в нашем приложении Tweeter. В этой точке им будет отображаться страница подтверждения — вы можете использовать страницу подтверждения, предлагаемую Passport по умолчанию, выполнив следующую команду:

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

Эта команда опубликует представление в файле `resources/views/vendor/passport/authorize.blade.php`, и ваши пользователи увидят страницу, показанную на рис. 13.1.

После того как пользователь подтвердит или отклонит авторизацию, Passport перенаправит его назад на предоставленный адрес `redirect_uri`. В примере 13.31 адрес `redirect_uri` был задан равным `url('tweeter/callback')`, поэтому пользователь перенаправляется назад на адрес `http://spacebook.test/tweeter/callback`.

Теперь запрос подтверждения будет содержать код, который может использоваться маршрутом обратного вызова приложения-потребителя для возвращения токена из приложения Tweeter с активизированным пакетом Passport. Запрос отклонения будет содержать сообщение об ошибке. Как может выглядеть маршрут обратного вызова приложения SpaceBook, показано в примере 13.32.

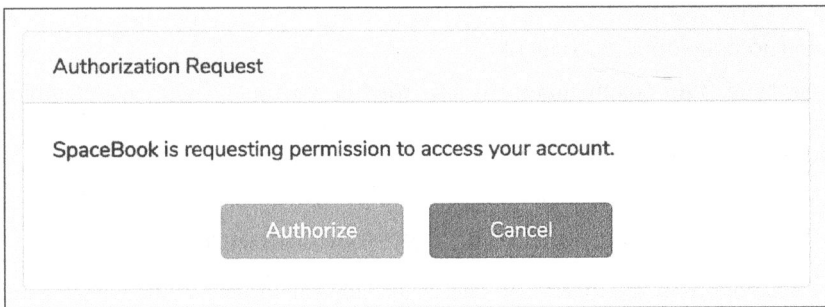


Рис. 13.1. Страница подтверждения кода авторизации OAuth

Пример 13.32. Пример маршрута обратного вызова авторизации в приложении-потребителе

```
// В файле routes/web.php приложения SpaceBook:
Route::get('tweeter/callback', function (Request $request) {
    if ($request->has('error')) {
        // Обработка состояния ошибки
    }

    $http = new GuzzleHttp\Client;

    $response = $http->post('http://tweeter.test/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => config('tweeter.id'),
            'client_secret' => config('tweeter.secret'),
            'redirect_uri' => url('tweeter/callback'),
            'code' => $request->code,
        ],
    ]);

    $thisUsersTokens = json_decode((string) $response->getBody(), true);
    // Выполнение определенных действий над токенами
});
```

Здесь разработчик приложения SpaceBook выполняет запрос с помощью библиотеки Guzzle HTTP к маршруту пакета Passport `/oauth/token` в приложении Tweeter. Затем он посылает POST-запрос с кодом авторизации, полученным при подтверждении пользователем доступа, и приложение Tweeter возвращает ответ в формате JSON, содержащий несколько ключей.

- ❑ **access_token.** Токен, который приложению SpaceBook потребуется сохранить для данного пользователя. С его помощью происходит аутентификация в последующих запросах к приложению Tweeter (используя заголовок `Authorization`).
- ❑ **refresh_token.** Токен, который потребуется приложению SpaceBook, если вы ограничите срок действия токенов. По умолчанию срок действия токенов доступа пакета Passport — один год.

- ❑ `expires_in`. Количество секунд до истечения срока действия токена `access_token` (должно обновляться).
- ❑ `token_type`. Тип возвращаемого вам токена, который будет выступать в роли токена `Bearer`. То есть во всех последующих запросах будет передаваться заголовок `Authorization` со значением `Bearer ВАШТОКЕН`.

ИСПОЛЬЗОВАНИЕ ТОКЕНОВ ОБНОВЛЕНИЯ

Если нужно, чтобы пользователи чаще проходили повторную аутентификацию, то следует задать более короткое время обновления токенов, после чего можно, используя токен обновления `refresh_token`, запрашивать новый токен доступа `access_token`, когда в этом возникает необходимость — обычно, когда вызов API возвращает код состояния 401 (`Unauthorized (Не авторизован)`).

Как задать более короткое время обновления, показано в примере 13.33.

Пример 13.33. Определение срока обновления токена

```
// Метод boot() провайдера AuthServiceProvider
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    // Срок действия токена до его обновления
    Passport::tokensExpireIn(
        now()->addDays(15)
    );

    // Срок действия токена до повторной аутентификации
    Passport::refreshTokensExpireIn(
        now()->addDays(30)
    );
}
```

Чтобы можно было запросить новый токен с помощью токена обновления, приложение-потребитель должно сохранить первый токен обновления `refresh_token` из изначального ответа аутентификации, представленного в примере 13.32. При обновлении приложение будет выполнять почти такой же вызов, как в примере 13.32, но слегка измененный, как показано в примере 13.34.

Пример 13.34. Запрашивание нового токена с помощью токена обновления

```
// В файле routes/web.php приложения SpaceBook:
Route::get('tweeter/request-refresh', function (Request $request) {
    $http = new GuzzleHttpClient;

    $params = [
        'grant_type' => 'refresh_token',
        'client_id' => config('tweeter.id'),
```

```
'client_secret' => config('tweeter.secret'),
'redirect_uri' => url('tweeter/callback'),
'refresh_token' => $theTokenYouSavedEarlier,
'scope' => '',
];

$response = $http->post(
    'http://tweeter.test/oauth/token',
    ['form_params' => $params]
);

$thisUsersTokens = json_decode(
    (string) $response->getBody(),
    true
);

// Выполнение определенных действий над токенами
});
```

В ответе приложение-потребитель получит новый набор токенов, чтобы сохранить его в своем объекте user.

Теперь у вас есть все необходимые инструменты для реализации простейших схем допуска по коду авторизации. Чуть позже мы рассмотрим, как создать панель администратора для своих клиентов и токенов, но сначала кратко обсудим другие типы допуска.

Токены личного доступа. Допуск по коду авторизации отлично подходит для приложений-потребителей, а допуск по паролю — для ваших собственных приложений. Но если разработчики приложения-потребителя захотят создать токены самостоятельно, чтобы проверить ваш API или использовать их на этапе разработки своего приложения? Как раз для этого нужны токены личного доступа.



Создание клиента с личным доступом

Для создания токенов личного доступа в вашей базе данных должен присутствовать клиент с личным доступом. Такой клиент уже будет добавлен туда при выполнении команды `php artisan passport:install`, но если по какой-либо причине нужно сгенерировать новый клиент с личным доступом, это можно сделать командой `php artisan passport:client --personal`:

```
$php artisan passport:client --personal
```

```
What should we name the personal access client?
[My Application Personal Access Client]:
> My Application Personal Access Client
```

```
Personal access client created successfully.
```

Строго говоря, токены личного доступа не тип «допуска». То есть они не основаны на предписываемой протоколом OAuth схеме аутентификации. Это предлагаемый пакетом Passport вспомогательный метод для легкой регистрации в системе одного клиента, единственным назначением которого является упрощение процесса создания вспомогательных токенов для разработчиков приложений-потребителей.

Допустим, что один из ваших пользователей разрабатывает аналог приложения SpaceBook для любителей марафонского бега с названием RaceBook. И ему нужно немного «поиграться» с API приложения Tweeter, чтобы понять принцип работы до написания кода. Имеет ли такой разработчик в своем распоряжении какой-либо инструмент для создания токенов, использующих схему допуска по коду авторизации? Пока нет, ведь у него вообще нет кода! Для этого и предназначены токены личного доступа.

Чтобы создать токены личного доступа, можно использовать API на базе JSON (см. далее), но вы также можете написать такой токен для пользователя непосредственно в коде:

```
// Создание токена без областей видимости
$token = $user->createToken('Token Name')->accessToken;
```

```
// Создание токена с областями видимости
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Ваши пользователи могут использовать эти токены точно так же, как токены, созданные с применением схемы допуска по коду авторизации. Области видимости рассмотрим в подразделе «Области видимости пакета Passport» на с. 384.

Токены от аутентификации сессии Laravel (токены синхронизатора)

Последний из возможных способов получения пользователями токенов для доступа к вашему API представляет собой еще один вспомогательный метод, предлагаемый пакетом Passport, но отсутствующий у обычных серверов OAuth. Этот метод предназначен для случая, когда пользователи прошли аутентификацию путем обычного входа в систему в вашем приложении Laravel и вам нужно обеспечить доступ к API для JavaScript-кода вашего приложения. Поскольку в таком случае обременительно повторно аутентифицировать пользователей по схеме допуска по коду авторизации или паролю, Laravel предлагает вспомогательный метод.

Если вы добавите middleware `Laravel\Passport\Http\Middleware\CreateFreshApiToken` в группу `web` (в файле `app/Http/Kernel.php`), то к каждому отправляемому фреймворком Laravel аутентифицированным пользователям ответу будет прикрепляться cookie-файл с именем `laravel_token`. Это веб-токен JSON (JSON Web Token, JWT) с закодированной информацией о CSRF-токене. Если при этом вы будете отправлять обычный CSRF-токен в своих запросах из JavaScript-кода в заголовке `X-CSRF-TOKEN`, а также отсылать заголовок `X-Requested-With` в любых

выполняемых вами запросах к API, то этот интерфейс будет сравнивать ваш CSRF-токен с cookie-файлом. И тем самым аутентифицировать пользователей для доступа к API, как в случае применения любого токена.

ВЕБ-ТОКЕНЫ JSON (JWT)

Токены JWT — это сравнительно новый формат «безопасного представления заявлений, направляемых друг другу двумя сторонами», который получил широкое распространение за последние несколько лет. Веб-токен JSON представляет собой объект JSON, содержащий всю необходимую информацию для определения статуса аутентификации и прав доступа пользователя. Этот объект JSON снабжается цифровой подписью с использованием хеш-кода аутентификации сообщения (HMAC) или алгоритма RSA, что подтверждает его надежность.

Данный токен обычно кодируется и затем доставляется посредством URL-адреса, POST-запроса или заголовка. После того как пользователь каким-либо образом аутентифицируется для доступа к системе, в каждый последующий HTTP-запрос включается токен с идентификационными и авторизационными данными пользователя.

Веб-токены JSON состоят из трех разделенных точками (.) строк в кодировке Base64, наподобие `xxx.yyy.zzz`. Первая секция — это представленный в кодировке Base64 JSON-объект с информацией об используемом алгоритме хеширования. Вторая — ряд «заявлений» в отношении авторизации и идентичности пользователя; а третья — подпись либо первая и вторая секции, зашифрованные и подписанные с использованием алгоритма, указанного в первой секции.

Более подробные сведения о токенах JWT можно найти на сайте [JWT.IO](http://jwt.io) по адресу <https://jwt.io/> или в документации пакета `Laravel: jwt-auth` (<http://bit.ly/2U6Uxf4>).

Данный заголовок уже определен в конфигурации начальной загрузки JavaScript, предлагаемой Laravel по умолчанию. Однако при использовании другого JavaScript-фреймворка потребуется определить его самостоятельно. В примере 13.35 показано, как это можно сделать в случае использования JQuery.

Пример 13.35. Настройка JQuery на передачу CSRF-токенов Laravel и заголовка `X-Requested-With` во всех AJAX-запросах

```
$.ajaxSetup({
  headers: {
    'X-CSRF-TOKEN': '{{ csrf_token() }}',
    'X-Requested-With': 'XMLHttpRequest'
  }
});
```

Если вы добавите middleware `CreateFreshApiToken` в группу `web` и будете передавать эти заголовки в каждом запросе из JavaScript-кода, то эти JavaScript-запросы смогут обращаться к защищенным пакетом Passport маршрутам API без всех сложностей допуска по паролю/коду авторизации.

Управление клиентами и токенами с помощью API пакета Passport и компонентов Vue

Мы знаем, как выполняется ручное создание клиентов и токенов и авторизация в качестве потребителя. Теперь коснемся части API пакета Passport, которая позволяет создавать элементы пользовательского интерфейса, предоставляющие пользователям возможность управлять своими клиентами и токенами.

Маршруты. Самый простой способ изучения маршрутов API — посмотреть, как работают и какие маршруты используют предоставленные в качестве образца компоненты Vue. Поэтому я приведу здесь их общий перечень:

- ❑ `/oauth/clients` (GET, POST);
- ❑ `/oauth/clients/{id}` (DELETE, PUT);
- ❑ `/oauth/personal-access-tokens` (GET, POST);
- ❑ `/oauth/personal-access-tokens/{id}` (DELETE);
- ❑ `/oauth/scopes` (GET);
- ❑ `/oauth/tokens` (GET);
- ❑ `/oauth/tokens/{id}` (DELETE).

Здесь несколько сущностей (клиенты, токены личного доступа, области видимости и токены). Мы можем перечислять все, создавать некоторые из них, а также удалять и обновлять (PUT) некоторые из них. Нельзя создавать области видимости, поскольку они определяются в коде, и токены, так как они создаются в схеме авторизации.

Компоненты Vue

Passport поставляется вместе с набором компонентов Vue, которые позволяют легко дать пользователям возможность управлять своими созданными клиентами, авторизованными клиентами (которым они предоставили доступ к своей учетной записи) и токенами личного доступа (для целей проводимого ими тестирования).

Чтобы опубликовать эти компоненты в своем приложении, выполните следующую команду:

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

После этого в файле `resources/js/components/passport` появятся три новых компонента Vue. Чтобы добавить их в начальную загрузку фреймворка Vue и сделать доступными в шаблонах, зарегистрируйте их в файле `resources/js/app.js`, как показано в примере 13.36.

Пример 13.36. Импорт Vue-компонентов пакета Passport в файл app.js

```
require('./bootstrap');

Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);

const app = new Vue({
  el: '#app'
});
```

После этого в любом месте приложения можно использовать следующие три компонента:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Компонент `<passport-clients>` отображает пользователю всех созданных им клиентов. Это означает, что разработчику приложения SpaceBook будет отображаться список клиентов SpaceBook, залогинившихся на сайте приложения Tweeter.

Компонент `<passport-authorized-clients>` отображает пользователю всех клиентов, которые прошли авторизацию для доступа к его учетной записи. Любой пользователь сразу двух приложений (SpaceBook и Tweeter), который предоставил SpaceBook доступ к своей учетной записи в Tweeter, увидит в этом списке приложение SpaceBook.

Компонент `<passport-personal-access-tokens>` отображает пользователю все созданные им токены личного доступа. Например, разработчик приложения RaceBook (аналог SpaceBook) увидит здесь токен личного доступа, используемый им для проверки API приложения Tweeter.

Если вы заново установили Laravel и хотите опробовать эти компоненты, выполните описанные ниже действия для их активизации.

- ☐ Выполните приведенные в этой главе инструкции по установке пакета Passport.
- ☐ Выполните в окне терминала следующие команды:

```
php artisan vendor:publish --tag=passport-components
npm install
npm run dev
php artisan make:auth
```

- ❑ Откройте файл `resources/views/home.blade.php` и добавьте ссылки для компонентов Vue (например, `<passport-clients></passport-clients>`) ниже элемента `<div class="card-body">`.

При желании можно использовать эти компоненты в неизменном виде. Однако они подходят в качестве примера применения API и для создания собственных клиентских компонентов в нужном вам формате.



Компиляция клиентских компонентов на базе Passport с помощью Laravel Elixir

Некоторые из описанных выше команд и инструкций будут выглядеть немного иначе в случае использования Laravel Elixir. Для получения подробной информации ознакомьтесь с документацией пакетов Passport (<http://bit.ly/2CFjF2y>) и Elixir (<http://bit.ly/2upTDf2>).

Области видимости пакета Passport

Мы практически не затрагивали области видимости. Для всего рассмотренного выше можно задать индивидуальную область видимости. Но перед этим кратко остановимся на том, что они собой представляют.

В OAuth области видимости — определенные наборы прав доступа, в чем-то отличные от полного разрешения на любые действия. Например, если вам приходилось получать токен API сервиса GitHub, то вы могли заметить, что одним приложениям нужен доступ только к вашему имени и адресу электронной почты, другим — ко всем вашим репозиториям, а третьим — к вашим «гистам». Все являются областью видимости, позволяющей пользователю и приложению-потребителю определить, в каком именно доступе нуждается приложение-потребитель для своей работы.

Как демонстрирует пример 13.37, можно определить области видимости для своего приложения в методе `boot()` провайдера `AuthServiceProvider`.

Пример 13.37. Определение областей видимости пакета Passport

```
// AuthServiceProvider
use Laravel\Passport\Passport;

...
public function boot()
{
    ...

    Passport::tokensCan([
        'list-clips' => 'List sound clips',
```



```
        'add-delete-clips' => 'Add new and delete old sound clips',
        'admin-account' => 'Administer account details',
    });
}
```

После того как вы определите свои области видимости, приложение-потребитель может запросить доступ к некоторым конкретным областям. Для этого нужно добавить разделенный пробелами список токенов в поле `scope` первоначального перенаправления, как показано в примере 13.38.

Пример 13.38. Запрашивание авторизации для доступа к конкретным областям видимости

```
// В файле routes/web.php приложения SpaceBook:
Route::get('tweeter/redirect', function () {
    $query = http_build_query([
        'client_id' => config('tweeter.id'),
        'redirect_uri' => url('tweeter/callback'),
        'response_type' => 'code',
        'scope' => 'list-clips add-delete-clips',
    ]);
    return redirect('http://tweeter.test/oauth/authorize?' . $query);
});
```

Когда пользователь попытается авторизовать данное приложение, оно предоставит список требуемых областей видимости. Таким образом, пользователь будет знать, что «SpaceBook запрашивает право на просмотр вашего адреса электронной почты» или «SpaceBook запрашивает право на публикацию записей от вашего имени, удаление ваших записей и отправку сообщений вашим друзьям».

Можно проверять области видимости с помощью хелпера или в экземпляре класса `User`. В примере 13.39 показано, как это можно делать во втором случае.

Пример 13.39. Проверка того, позволяет ли токен, с помощью которого аутентифицировался пользователь, выполнять определенное действие

```
Route::get('/events', function () {
    if (auth()->user()->tokenCan('add-delete-clips')) {
        //
    }
});
```

Такую проверку можно выполнять с помощью `middleware scope` и `scopes`. Добавьте их в массив `$routeMiddleware` в файле `app/Http/Kernel.php`:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

После этого можно использовать это `middleware` так, как в примере 13.40. Чтобы пользователь мог получить доступ к маршруту, `scopes` требует, чтобы в токене пользователя были указаны *все* определенные области видимости, а `scope` — чтобы была указана *как минимум одна* из определенных областей.

Пример 13.40. Использование middleware для ограничения доступа на основе областей видимости токенов

```
// routes/api.php
Route::get('clips', function () {
    // В токене доступа указаны обе области видимости
})->middleware('scopes:list-clips,add-delete-clips');

// или

Route::get('clips', function () {
    // В токене доступа указана минимум одна из перечисленных областей видимости
})->middleware('scope:list-clips,add-delete-clips');
```

Если вы не определили никаких областей видимости, то приложение будет работать так, как работало бы без них. Однако, как только вы начнете использовать области видимости, ваши приложения-потребители должны будут явно указывать, к каким конкретным областям они запрашивают доступ. Единственным исключением является использование допуска по паролю. В этом случае приложение-потребитель может запросить область видимости *, которая дает токену право на любые действия.

Развертывание пакета Passport

При первом развертывании приложения на базе Passport API пакета Passport не будет работать, пока для приложения не будут сгенерированы ключи. Это можно сделать, выполнив на эксплуатационном сервере команду `php artisan passport:keys`, которая сгенерирует ключи шифрования, используемые пакетом Passport при генерировании токенов.

Аутентификация с помощью токенов API

5.2 Laravel предлагает простой механизм аутентификации с помощью токенов API, который, по сути, мало чем отличается от использования имени пользователя и пароля. Каждому пользователю присваивается один токен, который может передаваться клиентами вместе с запросом для аутентификации этого запроса данного пользователя.

Этот механизм токенов API обеспечивает не столь надежную защиту, как протокол OAuth 2.0. Поэтому перед применением убедитесь, что он подойдет для вашего приложения. Поскольку у вас лишь один токен, то, как и в случае пароля, при его попадании в руки злоумышленника последний получает доступ ко всей системе. В то же время данный механизм безопаснее пароля потому, что вы можете использовать токены, труднее поддающиеся разгадыванию, а также выполнять очистку и сброс токенов при малейшем подозрении на несанкционированный доступ.

Таким образом, аутентификация с помощью токенов API может не подходить вашему приложению. Но если этот механизм все же подходит, его реализация не составит никаких проблем.

Сначала добавьте в таблицу `users` столбец `api_token` для 60-символьных уникальных значений:

```
$table->string('api_token', 60)->unique();
```

Затем отредактируйте метод для создания новых пользователей таким образом, чтобы он присваивал значение этому полю при образовании каждого нового пользователя. Это можно делать с помощью имеющегося в Laravel хелпера для генерирования случайных строк — тогда нужно каждый раз присваивать этому полю выражение `str_random(60)`. В случае добавления данного механизма в работающее приложение такое присваивание следует выполнить и уже существующим пользователям.

Для обертывания любых маршрутов в этот метод аутентификации используйте маршрутное middleware `auth:api`, как показано в примере 13.41.

Пример 13.41. Применение middleware API-аутентификации к группе маршрутов

```
Route::prefix('api')->middleware('auth:api')->group(function () {  
    //  
});
```

Поскольку вы используете нестандартный гард аутентификации, нужно его указывать при каждом применении любых методов `auth()`:

```
$user = auth()->guard('api')->user();
```

Настройка ответов с кодом 404

5.5 Laravel предлагает настраиваемые страницы сообщения об ошибке для обычных HTML-представлений. Однако вы можете настроить и предлагаемый по умолчанию резервный ответ с кодом 404 для вызовов с содержимым в формате JSON. Для этого добавьте в свой API вызов `Route::fallback()`, как показано в примере 13.42.

Пример 13.42. Определение резервного маршрута

```
// routes/api.php  
Route::fallback(function () {  
    return response()->json(['message' => 'Route Not Found'], 404);  
})->name('api.fallback.404');
```

5.5 **Активизация резервного маршрута.** Если нужно указать собственный маршрут в качестве маршрута, возвращаемого при перехвате фреймворком Laravel исключения «не найден», то отредактируйте обработчик исключений, используя метод `respondWithRoute()`, как показано в примере 13.43.

Пример 13.43. Вызов резервного маршрута при перехвате исключений «не найден»

```
// App\Exceptions\Handler
public function render($request, Exception $exception)
{
    if ($exception instanceof ModelNotFoundException && $request->isJson()) {
        return Route::respondWithRoute('api.fallback.404');
    }
    return parent::render($request, $exception);
}
```

Тестирование

Тестировать API в Laravel проще, чем почти все остальное.

Эта тема подробно рассмотрена в главе 12. Здесь лишь отмечу, что для этой цели подходит ряд методов для проверки утверждений в отношении формата JSON. В сочетании с простотой использования комплексных тестов приложения эта возможность позволяет быстро и легко делать свои тесты API. Как выглядит типичный шаблон тестирования API, показано в примере 13.44.

Пример 13.44. Типичный шаблон тестирования API

```
...
class DogsApiTest extends TestCase
{
    use WithoutMiddleware, RefreshDatabase;

    public function test_it_gets_all_dogs()
    {
        $dog1 = factory(Dog::class)->create();
        $dog2 = factory(Dog::class)->create();

        $response = $this->getJson('api/dogs');

        $response->assertJsonFragment(['name' => $dog1->name]);
        $response->assertJsonFragment(['name' => $dog2->name]);
    }
}
```

Чтобы не беспокоиться об аутентификации, мы используем трейт `WithoutMiddleware`. Если вам вообще это потребуется, тестирование аутентификации нужно будет выполнить отдельно (см. главу 9).

В этом тесте мы помещаем в БД два объекта `Dogs`, после чего переходим по маршруту API для перечисления всех объектов `Dogs` и убеждаемся, что оба объекта присутствуют в выходных данных.

Вы можете легко и просто охватить здесь все маршруты своего API, включая такие модифицирующие действия, как `POST` и `PATCH`.

Тестирование пакета Passport. Для тестирования своих областей видимости можно использовать метод `actingAs()` в фасаде `Passport`. В примере 13.45 показан типичный шаблон проверки областей видимости пакета `Passport`.

Пример 13.45. Тестирование доступа с использованием областей видимости

```
public function test_it_lists_all_clips_for_those_with_list_clips_scope()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['list-clips']
    );

    $response = $this->getJson('api/clips');
    $response->assertStatus(200);
}
```

Резюме

Laravel «заточен» под создание API и позволяет легко работать с API на базе JSON и RESTful. Фреймворк опирается на некоторые соглашения в отношении таких вещей, как разбивка на страницы. Однако по большей части вы сами решаете, какой способ сортировки, аутентификации и т. д. следует использовать для вашего API.

Laravel предоставляет инструменты для аутентификации и тестирования, простого манипулирования заголовками и их чтения, работы с форматом JSON и даже автоматического кодирования в формат JSON всех результатов Eloquent в случае, если они возвращаются маршрутом напрямую.

Laravel Passport представляет собой отдельный пакет, упрощающий создание и администрирование сервера OAuth в приложении Laravel.

14

Сохранение и извлечение данных

В главе 5 мы говорили, как сохраняются данные в реляционных БД. Существует еще много как локальных, так и удаленных способов. Здесь рассмотрим сохранение в файловой системе и оперативной памяти, загрузку файлов на сервер и манипулирование файлами, нереляционные хранилища данных, сессии, кэш, логирование, cookie-файлы и полнотекстовый поиск.

Локальные и облачные файловые менеджеры

Laravel предлагает ряд инструментов для манипулирования файлами через фасад `Storage`, а также несколько хелперов.

Инструменты для доступа к файловой системе Laravel позволяют подключаться к локальной файловой системе, а также к FTP-хранилищам, облачным хранилищам S3 и Rackspace. Файловые драйверы хранилищ S3 и Rackspace предоставляются пакетом `Flysystem` (<http://bit.ly/2upKDXr>). Можно легко добавлять в свое приложение Laravel дополнительные провайдеры пакета `Flysystem` — например, для облачных хранилищ `Dropbox` или `WebDAV`.

Настройка доступа к файлам

Определения файлового менеджера Laravel расположены в файле `config/file-systems.php`. Каждое подключение называется диском. Доступный по умолчанию список дисков показан в примере 14.1.

Пример 14.1. Доступные по умолчанию диски хранения

```
...  
'disks' => [  
    'local' => [  
        ...  
    ]  
]
```

```
'driver' => 'local',
'root' => storage_path('app'),
],

'public' => [
'driver' => 'local',
'root' => storage_path('app/public'),
'url' => env('APP_URL').'/storage',
'visibility' => 'public',
],

's3' => [
'driver' => 's3',
'key' => env('AWS_ACCESS_KEY_ID'),
'secret' => env('AWS_SECRET_ACCESS_KEY'),
'region' => env('AWS_DEFAULT_REGION'),
'bucket' => env('AWS_BUCKET'),
'url' => env('AWS_URL'),
],
],
```



Хелпер `storage_path()`

Используемый в примере 14.1 хелпер `storage_path()` предоставляет ссылку на каталог хранения, определенный в конфигурации Laravel, — `storage/`. Передаваемая хелперу строка размещается после имени этого каталога — так, вызов `storage_path('public')` возвратит строку `storage/public`.

Диск `local` обеспечивает подключение к локальной системе хранения и подразумевает взаимодействие с каталогом пути хранения `app`, то есть с каталогом `storage/app`.

Диск `public` также представляет собой локальный диск (при желании это можно изменить), предназначенный для раздаваемых вашим приложением файлов. По умолчанию расположен в каталоге `storage/app/public`. Чтобы использовать его для раздачи файлов, потребуется добавить символическую ссылку, указывающую внутрь каталога `public/`. К счастью, в нашем распоряжении есть команда Artisan, которая назначает путь `public/storage` в качестве пути для раздачи файлов из `storage/app/public`:

```
php artisan storage:link
```

Диск `s3` определяет способ подключения Laravel к облачным системам хранения файлов. Если вам приходилось подключаться к S3 или к любому другому провайдеру облачного хранилища, то вы уже знаете, как работать с этим диском: нужно передать ключ и пароль, а также некоторые сведения о нужной вам «папке» — в случае сервиса S3 это данные о регионе и корзине.

Использование фасада Storage

В файле `config/filesystem.php` вы можете задать «диск по умолчанию», который будет использоваться при каждом вызове фасада `Storage` без указания диска. Для указания диска нужно вызвать в фасаде метод `disk('diskname')`:

```
Storage::disk('s3')->get('file.jpg');
```

Все файловые системы предоставляют следующие методы.

- ❑ `get('file.jpg')`. Извлекает файл *file.jpg*.
- ❑ `put('file.jpg', $contentsOrStream)`. Помещает содержимое в файл *file.jpg*.
- ❑ `putFile('myDir', $file)`. Помещает содержимое указанного файла (в виде экземпляра класса `Illuminate\Http\File` или `Illuminate\Http\UploadedFile`) в каталог *myDir*, но с возложением контроля над всей потоковой обработкой и именованием файла на Laravel.
- ❑ `exists('file.jpg')`. Возвращает логическое значение, указывающее, существует ли *file.jpg*.
- ❑ `getVisibility('myPath')`. Получает статус видимости для указанного пути — `public` (публичный) или `private` (приватный).
- ❑ `setVisibility('myPath')`. Задаёт статус видимости для указанного пути — `public` или `private`.
- ❑ `copy('file.jpg', 'newfile.jpg')`. Копирует *file.jpg* в файл *newfile.jpg*.
- ❑ `move('file.jpg', 'newfile.jpg')`. Перемещает *file.jpg* в файл *newfile.jpg*.
- ❑ `prepend('my.Log', 'log text')`. Добавляет содержимое в начале файла *my.Log*.
- ❑ `append('my.Log', 'log text')`. Добавляет содержимое в конце *my.Log*.
- ❑ `delete('file.jpg')`. Удаляет *file.jpg*.
- ❑ `size('file.jpg')`. Возвращает размер файла *file.jpg* в байтах.
- ❑ `lastModified('file.jpg')`. Возвращает временную метку Unix для момента последнего изменения *file.jpg*.
- ❑ `files('myDir')`. Возвращает массив имен файлов, расположенных в каталоге *myDir*.
- ❑ `allFiles('myDir')`. Возвращает массив имен файлов, расположенных в *myDir* и во всех подкаталогах.
- ❑ `directories('myDir')`. Возвращает массив имен каталогов, расположенных в *myDir*.
- ❑ `allDirectories('myDir')`. Возвращает массив имен каталогов, расположенных в каталоге *myDir* и во всех подкаталогах.
- ❑ `makeDirectory('myDir')`. Создает новый каталог.
- ❑ `deleteDirectory('myDir')`. Удаляет *myDir*.



Внедрение экземпляра

Если хотите внедрять экземпляр, а не использовать фасад File, укажите в подсказках типов или внедрите класс `Illuminate\Filesystem\Filesystem` и получите в свое распоряжение все те же методы.

Добавление дополнительных провайдеров из пакета Flysystem

Для добавления дополнительного провайдера из пакета Flysystem потребуется «расширить» стандартную систему хранения Laravel. Где-нибудь в сервис-провайдере (например в методе `boot()` провайдера `AppServiceProvider`, но, возможно, правильнее создавать отдельный сервис-провайдер для каждой новой привязки)— добавьте новые системы хранения с помощью фасада `Storage`, как показано в примере 14.2.

Пример 14.2. Добавление дополнительных провайдеров из пакета Flysystem

```
// Сервис-провайдер
public function boot()
{
    Storage::extend('dropbox', function ($app, $config) {
        $client = new DropboxClient(
            $config['accessToken'], $config['clientIdentifier']
        );

        return new Filesystem(new DropboxAdapter($client));
    });
}
```

Базовые способы загрузки файлов на сервер и манипулирования файлами

Фасад `Storage` часто используется для принятия файлов, загружаемых на сервер пользователями приложения. Типичный способ решения этой задачи показан в примере 14.3.

Пример 14.3. Типичный способ реализации загрузки пользователями файлов на сервер

```
...
class DogsController
{
    public function updatePicture(Request $request, Dog $dog)
    {
        Storage::put(
            "dogs/{$dog->id}",
            file_get_contents($request->file('picture')->getRealPath())
        );
    }
}
```

Методом `put()` мы помещаем содержимое в файл `dogs/id`, получая его из загружаемого на сервер файла. Такой файл является потомком класса `SplFileInfo`, который предоставляет метод `getRealPath()`, возвращающий путь к месту размещения файла. Таким образом, мы получаем временный путь файла, загружаемого пользователем на сервер, считываем его с помощью метода `file_get_contents()` и передаем методу `Storage::put()`.

Поскольку при этом мы получаем доступ к файлу, можно произвести нужные нам манипуляции над файлом до его сохранения — изменить его размеры с помощью пакета для обработки изображений, если это графический файл, проверить на соответствие определенным критериям и т. д.

Если бы нужно было загружать этот же файл в хранилище S3, притом что наши учетные данные хранились бы в файле `config/filesystems.php`, то мы могли бы просто изменить вызов в примере 14.3 следующим образом: `Storage::disk('s3')->put()`. В примере 14.4 показана более сложная загрузка на сервер.

Пример 14.4. Более сложный пример загрузки файлов на сервер с использованием библиотеки Intervention

```
...
class DogsController
{
    public function updatePicture(Request $request, Dog $dog)
    {
        $original = $request->file('picture');

        // Изменение размера изображения до максимальной ширины 150
        $image = Image::make($original)->resize(150, null, function ($constraint) {
            $constraint->aspectRatio();
        })->encode('jpg', 75);

        Storage::put(
            "dogs/thumbs/{$dog->id}",
            $image->getEncoded()
        );
    }
}
```

В примере 14.4 я использовал библиотеку для обработки изображений с названием Intervention (<http://image.intervention.io>). Вы можете применить любую нужную вам библиотеку. Важный момент: можно свободно выполнять над файлами любые манипуляции до их сохранения.



Использование методов `store()` и `storeAs()` в объекте загружаемого на сервер файла



В Laravel 5.3 появилась возможность сохранять загружаемый на сервер файл, используя сам файл. Это показано в примере 7.12.

Простые способы скачивания файлов

Фасад **Storage** не только позволяет легко принимать загружаемые пользователями на сервер файлы, но и упрощает возвращение файлов пользователям. В простейшем случае это выглядит так, как в примере 14.5.

Пример 14.5. Простейшая реализация скачивания файлов

```
public function downloadMyFile()
{
    return Storage::download('my-file.pdf');
}
```

Сессии

Сохранение сессии — основной инструмент для сохранения состояния между запросами страниц. Менеджер сессий **Laravel** поддерживает драйверы сессии для работы с файлами, cookie-файлами, базами данных, хранилищами **Memcached** и **Redis**, и размещаемыми в оперативной памяти массивами. Последние хранятся лишь на протяжении обработки запроса страницы и подходят только для тестирования.

Вы можете настроить все параметры и драйверы сессии в файле `config/session.php`. Можно указать, следует ли шифровать данные сессии, какой драйвер использовать (по умолчанию идет драйвер `file`), а также задать более специфические параметры подключения — размер хранилища сессии, какие файлы или таблицы базы данных нужны и т. п. Чтобы узнать, какие именно зависимости и параметры нужны для выбранного драйвера, ознакомьтесь с документацией по сессиям (<http://bit.ly/2HFXsW7>).

Универсальный API инструментов сессии позволяет сохранять и извлекать данные по индивидуальному ключу: например, `session()->put('user_id')` и `session()->get('user_id')`. При этом ни в коем случае не сохраняйте что-либо по ключу сессии `flash`, поскольку он предназначен для внутреннего использования **Laravel** флеш-памятью сессии (доступной только для следующего запроса страницы).

Получение доступа к сессии

Наиболее широко используемый способ получения доступа к сессии сводится к применению фасада **Session**:

```
session()->get('user_id');
```

Однако подходит ли метод `session()` в любом объекте **Illuminate Request**, как в примере 14.6.

Пример 14.6. Использование метода `session()` в объекте `Request`

```
Route::get('dashboard', function (Request $request) {
    $request->session()->get('user_id');
});
```

Еще один способ — внедрить экземпляр класса `Illuminate\Session\Store`, как показано в примере 14.7.

Пример 14.7. Внедрение базового класса сессии

```
Route::get('dashboard', function (Illuminate\Session\Store $session) {
    return $session->get('user_id');
});
```

Наконец, можно воспользоваться глобальным хелпером `session()`. Как показано в примере 14.8, он применяется без параметров для получения экземпляра сессии: с одним строковым параметром — для извлечения данных из сессии и с массивом — для сохранения данных в сессию.

Пример 14.8. Использование глобального хелпера `session()`

```
// Извлечение данных
$value = session()->get('key');
$value = session('key');
// Сохранение данных
session()->put('key', 'value');
session(['key', 'value']);
```

Если вы еще только учитесь использовать `Laravel` и не уверены, какой способ лучше выбрать, я рекомендую вариант с глобальным хелпером.

Методы, доступные в экземплярах сессий

Наиболее часто используемые методы — `get()` и `put()`. Однако кратко рассмотрим все их доступные параметры.

- ❑ `session()->get($key, $fallbackValue)`. Метод `get()` извлекает из сессии значение указанного ключа. Если ему не присвоено значение, метод возвращает резервное (или значение `null`, если резервное не определено). Как показывают следующие примеры, в качестве резервного значения может выступать строка или замыкание:

```
$points = session()->get('points');

$points = session()->get('points', 0);

$points = session()->get('points', function () {
    return (new PointGetterService)->getPoints();
});
```

- ❑ `session()->put($key, $value)`. Метод `put()` сохраняет указанное значение в сессии по указанному ключу:

```
session()->put('points', 45);

$points = session()->get('points');
```

- ❑ `session()->push($key, $value)`. Если определенные значения сессии представляют собой массивы, то можно использовать метод `push()` для добавления значения в такой массив:

```
session()->put('friends', ['Saúl', 'Quang', 'Mechteld']);

session()->push('friends', 'Javier');
```

- ❑ `session()->has($key)`. Метод `has()` проверяет, задано ли значение с указанным ключом:

```
if (session()->has('points')) {
    // Выполнение некоторых действий
}
```

Вы также можете передать массив ключей. Тогда метод возвратит значение `true`, если заданы значения для всех указанных ключей.



Метод `session()->has()` и значение `null`

Если определенное значение сессии задано, но равняется `null`, метод `session()->has()` возвратит значение `false`.

- ❑ `session()->exists($key)`. Как и `has()`, метод `exists()` проверяет, задано ли значение с указанным ключом, но возвращает значение `true` даже в том случае, когда заданное значение равняется `null`:

```
if (session()->exists('points')) {
    // Возвращает значение true, даже если ключу points присвоено значение null
}
```

- ❑ `session()->all()`. Метод `all()` возвращает массив со всеми значениями сессии, включая заданные фреймворком. Кроме того, вы увидите здесь значения с такими ключами, как `_token` (CSRF-токен), `_previous` (предыдущая страница, для перенаправлений `back()`) и `flash` (флеш-память).

- ❑ `session()->forget($key)` и `session()->flush()`. Метод `forget()` удаляет заданное ранее значение сессии. Метод `flush()` удаляет все значения сессии, включая заданные фреймворком:

```
session()->put('a', 'awesome');
session()->put('b', 'bodacious');
```

```
session()->forget('a');  
// Значения с ключом а уже нет; значение с ключом b по-прежнему существует  
session()->flush();  
// Сессия очищена
```

- ❑ `session()->pull($key, $fallbackValue)`. Как и `get()`, метод `pull()` извлекает из сессии значение, но при этом оно удаляется из сессии после извлечения.
- ❑ `session()->regenerate()`. Иногда нужно сгенерировать новый идентификатор сессии. Это можно сделать с помощью метода `regenerate()`.

Флеш-память сессии

Осталось рассмотреть еще три метода, имеющих отношение к так называемой флеш-памяти сессии.

При сохранении сессии нередко требуется задавать значение таким образом, чтобы оно было доступным только при загрузке следующей страницы. Например, иногда нужно сохранить строку, сообщающую что-то вроде «Запись успешно обновлена». Конечно, такое сообщение можно получать вручную, а затем стирать его при загрузке следующей страницы. Но при частом применении такой подход будет требовать слишком больших сил. Здесь в игру вступает флеш-память сессии: ключи, срок жизни которых ограничен лишь одним запросом страницы.

При этом все необходимое делает Laravel — вам остается лишь воспользоваться `flash()` вместо `put()`. Таким образом, для работы с флеш-памятью можно использовать следующие методы:

- ❑ `session()->flash($key, $value)`. Метод `flash()` присваивает ключу сессии указанное значение, которое будет доступно только для следующего запроса страницы.
- ❑ `session()->reflash()` и `session()->keep($key)`. Если нужно, чтобы флеш-данные сессии предыдущей страницы оставались в наличии в течение еще одного запроса, то можно пересохранить все эти данные для следующего запроса с помощью метода `reflash()`. Либо методом `keep($key)` пересохранить для следующего запроса только одно флеш-значение. `keep()` также позволяет пересохранять несколько значений путем передачи ему массива ключей.

Кэш

Кэши по своей структуре очень сходны с сессиями. Вы предоставляете значение определенного ключа, и Laravel его сохраняет. Основное различие в том, что в первом случае кэшируются данные отдельного приложения, а во втором — отдельного пользователя. Это означает, что кэши обычно используются для сохранения результатов запросов к базе данных, вызовов API или других

медленно выполняемых запросов, которые допускают определенную степень «несвежести».

Конфигурационные параметры кэша доступны в файле `config/cache.php`. Так же как и в случае сессий, можно задать конкретные параметры конфигурации для любых применяемых драйверов, а также указать драйвер по умолчанию. По умолчанию Laravel использует драйвер кэша `file`, но подходит и драйвер для хранилища `Memcached`, `Redis` или `APC`, базы данных или собственный драйвер кэша. Узнать, какие именно зависимости и параметры необходимо приготовить для этого, можно в документации по использованию кэша (<http://bit.ly/2Yk60qV>).



Определение длительности хранения кэша в минутах или секундах

До версии Laravel 5.8 при передаче любому методу для работы с кэшем целого числа в качестве параметра, определяющего длительность хранения кэша, это число представляло количество минут, в течение которых нужно было сохранять в кэше тот или иной элемент. В версии 5.8 и следующих это число представляет количество секунд.

Получение доступа к кэшу

Как и в случае сессий, существует несколько способов доступа к кэшу. Вы можете использовать фасад:

```
$users = Cache::get('users');
```

Можно также получать экземпляры из контейнера, как в примере 14.9.

Пример 14.9. Внедрение экземпляра кэша

```
Route::get('users', function (Illuminate\Contracts\Cache\Repository $cache) {
    return $cache->get('users');
});
```

Еще один способ — использовать глобальный хелпер `cache()` (появился в версии Laravel 5.3) (пример 14.10).

Пример 14.10. Использование глобального хелпера `cache()`

```
// Извлечение данных из кэша
$users = cache('key', 'default value');
$users = cache()->get('key', 'default value');
// Сохранение данных в течение количества секунд, указанного в переменной $seconds
$users = cache(['key' => 'value'], $seconds);
$users = cache()->put('key', 'value', $seconds);
```

Если вы не слишком хорошо знакомы с Laravel и не знаете, какой способ лучше, я рекомендую использовать глобальный хелпер.

Методы, доступные в экземплярах кэшей

Рассмотрим некоторые методы, которые можно вызывать в объекте `Cache`.

- ❑ `cache()->get($key, $fallbackValue)` и `cache()->pull($key, $fallbackValue)`. Метод `get()` позволяет легко извлекать значение любого указанного ключа. `pull()` аналогичен, только удаляет значение из кэша после его извлечения.

- ❑ `cache()->put($key, $value, $secondsOrExpiration)`. Метод `put()` задает значение указанного ключа в секундах. Если вместо длительности хранения в секундах вы предпочитаете задать дату и время истечения срока хранения, можно передать объект `Carbon` в качестве третьего параметра:

```
cache()->put('key', 'value', now()->addDay());
```

- ❑ `cache()->add($key, $value)`. Метод `add()` аналогичен `put()`, но не позволяет задать уже существующее значение. Он также возвращает логическое значение, указывающее, было произведено добавление или нет.

```
$someDate = now();
cache()->add('someDate', $someDate); // возвращает true
$someOtherDate = now()->addHour();
cache()->add('someDate', $someOtherDate); // возвращает false
```

- ❑ `cache()->forever($key, $value)`. Метод `forever()` сохраняет в кэш значение указанного ключа. В отличие от `put()`, оно хранится бесконечно долго (пока не будет удалено с помощью `forget()`).

- ❑ `cache()->has($key)`. Метод `has()` возвращает логическое значение, указывающее, существует или нет значение с указанным ключом.

- ❑ `cache()->remember($key, $seconds, $closure)` и `cache()->rememberForever($key, $closure)`. Метод `remember()` позволяет реализовать с помощью одного метода очень распространенный сценарий обработки. При этом требуется проверить, существует ли в кэше значение с указанным ключом, и, если нет, каким-то образом получить его, сохранить в кэше и возвратить. `remember()` может передать проверяемый ключ, длительность хранения в секундах, а также замыкание, определяющее способ получения значения в случае отсутствия значения с указанным ключом. Метод `rememberForever()` аналогичен, только значение хранится бесконечно долго. Соответственно, не нужно указывать длительность хранения в секундах. Следующий пример демонстрирует распространенный сценарий применения метода `remember()`:

```
// Либо извлекает из кэша значение ключа "users", либо получает результат
// метода User::all(), сохраняет его в кэше с ключом "users" и возвращает его
$users = cache()->remember('users', 7200, function () {
    return User::all();
});
```

- ❑ `cache()->increment($key, $amount)` и `cache()->decrement($key, $amount)`. Методы `increment()` и `decrement()` позволяют увеличивать и уменьшать на единицу содержащиеся в кэше целочисленные значения. Если в кэше нет значения с указанным ключом, то исходное значение считается равным 0. Кроме того, можно

увеличивать и уменьшать значение на некоторое другое значение, передав его в качестве второго параметра.

- ❑ `cache()->forget($key)` и `cache()->flush()`. Метод `forget()` аналогичен `forget()` фасада `Session`: стирает значение указанного ключа. Метод `flush()` стирает все содержимое кэша.

Cookie-файлы

Работа с cookie-файлами осуществляется так же, как с сессиями и кэшем. Для них тоже предусмотрен специальный фасад и глобальный хелпер, здесь мы руководствуемся той же моделью: нам нужно просто извлекать и задавать значения.

Однако в силу того, что cookie-файлы неизбежно привязаны к запросам и ответам, с ними потребуется взаимодействовать по-другому. Кратко рассмотрим различия.

Cookie-файлы в Laravel

В Laravel cookie-файлы могут находиться в трех местах. Они могут поступать вместе с запросом. Тогда в момент перехода на страницу пользователь уже имеет cookie-файл, который можно считать с помощью фасада `Cookie` или в объекте запроса.

Они также могут быть отправлены вместе с ответом. В таком случае ответ дает указание браузеру пользователя сохранить cookie-файл для последующих посещений страницы. Это реализуется добавлением cookie-файла в свой объект ответа перед его возвращением.

И наконец, cookie-файлы могут быть *в очереди*. При установке cookie-файла с помощью фасада `Cookie` он помещается в очередь `CookieJar`, а затем удаляется оттуда и добавляется в объект ответа с помощью middleware `AddQueuedCookiesToResponse`.

Получение доступа к cookie-файлам

Есть три способа чтения и установки cookie-файлов: в фасаде `Cookie`, с помощью глобального хелпера `cookie()` и в объектах запроса и ответа.

Фасад `Cookie`

Фасад `Cookie` предоставляет наиболее широкие возможности, позволяя не только считывать и создавать cookie-файлы, но и помещать их в очередь на добавление в ответ. Он предоставляет следующие методы.

- ❑ `Cookie::get($key)`. Чтобы извлечь значение cookie-файла, поступившего вместе с запросом, можно выполнить вызов `Cookie::get('имя-cookie-файла')`. Это самый простой способ.

- ❑ `Cookie::has($key)`. Вы можете проверить, поступил ли cookie-файл вместе с запросом, выполнив вызов `Cookie::has('имя-cookie-файла')`, который возвратит логическое значение.
- ❑ `Cookie::make(...параметры)`. Если нужно *создать* cookie-файл без помещения его в какую-либо очередь, вызовите метод `Cookie::make()`. Наиболее вероятный сценарий его использования сводится к тому, чтобы сначала создать cookie-файл, а затем вручную прикрепить его к объекту ответа. Это рассмотрим чуть позже.

Параметры метода `make()` в порядке их следования включают в себя:

- `$name` — имя cookie-файла;
- `$value` — содержимое cookie-файла;
- `$minutes` — срок хранения cookie-файла;
- `$path` — путь, для которого будет действителен cookie-файл;
- `$domain` — список доменов, для которых должен использоваться cookie-файл;
- `$secure` — указывает, должен ли cookie-файл передаваться только через защищенное соединение (HTTPS);
- `$httpOnly` — указывает, будет ли cookie-файл доступен только по протоколу HTTP;
- `$raw` — указывает, должен ли cookie-файл отправляться без кодирования URL-адреса;
- `$sameSite` — указывает, следует ли предоставлять доступ к cookie-файлу для межсайтовых запросов; возможны варианты `lax`, `strict` и `null`;
- `Cookie::make()` — возвращает экземпляр класса `Symfony\Component\HttpFoundation\Cookie`.



Настройки по умолчанию для cookie-файлов

Очередь `CookieJar`, используемая экземпляром фасада `Cookie`, считывает свои значения по умолчанию из конфигурации сессии. Таким образом, при изменении любых конфигурационных значений для cookie-файлов сессии в файле `config/session.php` эти значения по умолчанию будут применяться ко всем cookie-файлам, создаваемым с помощью `Cookie`.

- ❑ `Cookie::queue(cookie-файл || параметры)`. При использовании метода `Cookie::make()` дополнительно потребуется прикрепить cookie-файл к своему ответу, о чем мы поговорим чуть позже. Метод `Cookie::queue()` предлагает тот же синтаксис, что и метод `Cookie::make()`, но при этом созданный cookie-файл помещается в очередь на автоматическое прикрепление к ответу с помощью `middleware`. При желании вы можете просто передавать созданный вами cookie-файл методу.

- ❑ `Cookie::queue()`. Самый простой способ добавления cookie-файла в ответ в Laravel выглядит следующим образом:

```
Cookie::queue('dismissed-popup', true, 15);
```



Когда помещенные в очередь cookie-файлы не устанавливаются

Cookie-файлы могут возвращаться только вместе с ответом. Поэтому если после внесения cookie-файла в очередь с помощью фасада `Cookie` ваш ответ не будет возвращен должным образом (например, будет вызван метод `exit()` языка PHP или что-либо прервет выполнение вашего сценария), то ваш cookie-файл не будет установлен.

Глобальный хелпер `cookie()`

При вызове без параметров глобальный хелпер `cookie()` возвращает объект `CookieJar`. Однако два очень удобных метода фасада `Cookie` — `has()` и `get()` — доступны *только* в фасаде, отсутствуя в `CookieJar`. На мой взгляд, в этом контексте глобальный хелпер не так удобен, как остальные варианты.

Глобальная вспомогательная функция `cookie()` подходит для создания cookie-файла. Когда вы передаете параметры функции `cookie()`, они отправляются непосредственно эквиваленту функции `Cookie::make()`, что делает это самым быстрым способом создания cookie-файла:

```
$cookie = cookie('dismissed-popup', true, 15);
```



Внедрение экземпляра

Вы также можете внедрить экземпляр класса `Illuminate\Cookie\CookieJar` в любом месте приложения, но при этом получите такие же ограничения, о каких говорится здесь.

Работа с cookie-файлами в объектах `Request` и `Response`

Поскольку cookie-файлы поступают в составе запроса и устанавливаются в ответе, именно эти объекты `Illuminate` являются местом их реального размещения. Методы `get()`, `has()` и `queue()` фасада `Cookie` на самом деле лишь `middleware`, взаимодействующее с объектами `Request` и `Response`.

Соответственно, самый простой способ взаимодействия с cookie-файлами — извлекать из запроса и устанавливать в ответе.

Чтение cookie-файлов из объектов `Request`. Получив в свое распоряжение объект `Request` (воспользуйтесь вызовом вида `app('request')`), вы можете считать

cookie-файлы этого объекта `Request`, вызвав в нем метод `cookie()`, как показано в примере 14.11.

Пример 14.11. Чтение cookie-файлов из объекта `Request`

```
Route::get('dashboard', function (Illuminate\Http\Request $request) {
    $userDismissedPopup = $request->cookie('dismissed-popup', false);
});
```

Метод `cookie()` принимает два параметра: имя cookie-файла и опционально резервное значение.

Установка cookie-файлов в объектах `Response`. Имея в наличии готовый объект `Response`, можно добавить в него cookie-файлы, вызвав в нем метод `cookie()` (или `withCookie()` в Laravel до версии 5.3), как в примере 14.12.

Пример 14.12. Установка cookie-файла в объекте `Response`

```
Route::get('dashboard', function () {
    $cookie = cookie('saw-dashboard', true);

    return Response::view('dashboard')
        ->cookie($cookie);
});
```

Если вы еще не слишком хорошо знакомы с Laravel и не знаете, какой способ лучше выбрать, я рекомендую работать с cookie-файлами в объектах `Request` и `Response`. Это чуть сложнее, зато позволит избежать лишних сюрпризов, если следующий разработчик не будет знаком с очередью `CookieJar`.

Логирование

Вы уже видели несколько примеров логирования (`logging`), когда мы обсуждали контейнер и фасады. Будет нелишним кратко остановиться на том, какие возможности логирования вам доступны, помимо простого вызова вида `Log::info('Message')`.

Логирование выполняется с целью повышения «открытости» кода, то есть вашей способности понять, что происходит в приложении в нужный момент.

Логи — короткие сообщения, иногда с определенной информацией в удобной для человека форме, генерируемые кодом, чтобы вам было проще понять, что происходило во время выполнения приложения. Каждый лог перехватывается на определенном *уровне*, который может варьироваться от `emergency` (аварийный режим — для потенциально опасных событий) до `debug` (отладка — для малозначительных событий).

По умолчанию приложение записывает любые логи в файл `storage/logs/laravel.log`, и при этом каждое сообщение выглядит примерно следующим образом:

```
[2018-09-22 21:34:38] local.ERROR: Something went wrong.
```

Здесь есть дата и время, тип используемой среды, уровень ошибки и текст сообщения — все в одной строке. В то же время Laravel по умолчанию логирует все неперехваченные исключения, и тогда в строке выводится полная трассировка стека.

В следующем разделе мы поговорим о том, как выполняется логирование, зачем это необходимо и как можно использовать нестандартные способы логирования (например, с применением Slack-канала).

Когда и зачем следует выполнять логирование

Чаще всего логирование выполняется для предоставления частично очищаемой ведомости событий, которые *могут* быть важны в дальнейшем, но не требуют с абсолютной однозначностью программного доступа к ним. Логирование скорее нужно для информирования вас о событиях в приложении, чем для создания используемых в приложении структурированных данных.

Если нужно, чтобы какой-то код принимал запись о каждом входе пользователя в систему и производил над ней некоторые действия, то лучше использовать базу данных для *событий* входа в систему. Однако если эти события входа в систему требуют вашего внимания, но у вас нет уверенности в необходимости этой информации и в том, нужен ли программный доступ к ней, можно выдавать соответствующее сообщение лога уровня `debug` (отладка) или `info` (справочная информация), и спать спокойно.

Логирование также часто применяется, когда требуется узнать значения определенного кода в момент появления ошибки, или в определенное время суток, или в любой другой ситуации, подразумевающей необходимость получения данных, когда вас нет поблизости. Добавьте лог-инструкции в коде, получите необходимые данные из логов и сохраните их в коде для последующего использования или просто удалите.

Внесение записей в логи

Записать лог в Laravel проще всего с помощью фасада `Log`. При этом следует вызывать в этом фасаде метод, подходящий уровню серьезности сообщения. Доступные уровни соответствуют определению уровней в стандарте RFC 5424 (<http://bit.ly/2YltbAS>):

```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

В качестве необязательного второго параметра также можно передать массив сопутствующих данных:

```
Log::error('Failed to upload user image.', ['user' => $user]);
```

Эта дополнительная информация может перехватываться в зависимости от используемого лога. Так она будет выглядеть в локальном логе по умолчанию (с тем отличием, что в логе это будет только одна строка):

```
[2018-09-27 20:53:31] local.ERROR: Failed to upload user image. {
  "user": "[object] (App\\User: {
    \"id\":1,
    \"name\": \"Matt\",
    \"email\": \"matt@tighten.co\",
    \"email_verified_at\":null,
    \"api_token\": \"long-token-here\",
    \"created_at\": \"2018-09-22 21:39:55\",
    \"updated_at\": \"2018-09-22 21:40:08\"
  })"
}
```

Каналы логирования

5.6 В Laravel 5.6 был значительно изменен способ настройки и перехватывания логов с введением концепции различных *каналов* и *драйверов*. Если вы работаете с версией 5.5 или более ранней, можно пропустить данный материал и сразу перейти к разделу «Полнотекстовый поиск с использованием Laravel Scout» на с. 409.

Как и в случае таких функций Laravel, как сохранение файлов, работа с базами данных, электронной почтой и т. д., логирование можно настроить на один/несколько предопределенных типов логов, указав их в файле конфигурации. Использование каждого типа подразумевает передачу своего набора конфигурационных параметров соответствующему драйверу логирования.

Эти типы логирования называют каналами. По умолчанию вам доступны каналы **stack**, **single**, **daily**, **slack**, **stderr**, **syslog** и **errorlog**. Каждый подключается к отдельному драйверу — **single**, **daily**, **slack**, **syslog**, **errorlog**, **monolog**, **custom** и **stack**.

Здесь мы рассмотрим самые популярные каналы: **stack**, **single**, **daily** и **slack**. Подробные сведения о драйверах и полный список доступных каналов смотрите в документации по логированию (<http://bit.ly/2TVgSWT>).

Канал single

Канал **single** записывает каждый лог в отдельный файл, путь к которому задается в ключе **path**. В примере 14.13 показана предлагаемая по умолчанию конфигурация этого канала.

Пример 14.13. Конфигурация по умолчанию для канала `single`

```
'single' => [  
    'driver' => 'single',  
    'path' => storage_path('logs/laravel.log'),  
    'level' => 'debug',  
],
```

Иначе говоря, данный канал логирует события только `debug` или более высокого уровня, внося записи в отдельный файл `storage/logs/laravel.log`.

Канал `daily`

Канал `daily` (ежедневный) выделяет новый файл для каждого нового дня. Конфигурация по умолчанию показана в примере 14.14.

Пример 14.14. Конфигурация по умолчанию для канала `daily`

```
'daily' => [  
    'driver' => 'daily',  
    'path' => storage_path('logs/laravel.log'),  
    'level' => 'debug',  
    'days' => 7,  
],
```

Данный канал сходен с `single`, но здесь можно указать срок хранения логов в днях до их очистки, а к указываемому нами имени файла добавляется дата. Так, представленная выше конфигурация сгенерирует файл с названием `storage/logs/laravel-{гггг-мм-дд}.log`.

Канал `slack`

Канал `slack` позволяет легко отправлять в Slack свои логи (выборочным образом).

Данный канал также показывает, что вы не ограничены в возможностях только тех обработчиков, которые предоставляются вместе с Laravel. Позже мы коснемся этого подробнее, а пока отмечу, что здесь мы не имеем дело с пользовательской реализацией работы с сервисом Slack. В данном случае Laravel предоставляет драйвер логирования, который подключается к обработчику Slack библиотеки Monolog. Если можно использовать любой обработчик библиотеки Monolog, значит, вам доступно и множество других вариантов.

Конфигурацию по умолчанию этого канала см. в примере 14.15.

Пример 14.15. Конфигурация по умолчанию для канала `slack`

```
'slack' => [  
    'driver' => 'slack',  
    'url' => env('LOG_SLACK_WEBHOOK_URL'),
```

```

    'username' => 'Laravel Log',
    'emoji' => ':boom:',
    'level' => 'critical',
],

```

Канал stack

Канал `stack` включен по умолчанию в вашем приложении. В примере 14.16 показано, как выглядит его конфигурация по умолчанию в версии 5.7 и следующих.

Пример 14.16. Конфигурация по умолчанию для канала `stack`

```

'stack' => [
    'driver' => 'stack',
    'channels' => ['daily'],
    'ignore_exceptions' => false,
],

```

Канал `stack` позволяет передавать все свои логи сразу в несколько каналов, указанных в массиве `channels`. Ваши приложения Laravel настраиваются на этот канал по умолчанию. В силу того что по умолчанию в версии 5.8 и более новых массив `channels` содержит только значение `daily`, в действительности приложение будет применять канал логирования `daily`.

Но если нужно, чтобы все логи `info` и более высокого уровня занолись в ежедневные файлы, а сообщения `critical` и более высокого уровня передавались в Slack? Это можно легко реализовать с помощью драйвера `stack`, как показано в примере 14.17.

Пример 14.17. Настройка драйвера `stack`

```

'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['daily', 'slack'],
    ],

    'daily' => [
        'driver' => 'daily',
        'path' => storage_path('logs/laravel.log'),
        'level' => 'info',
        'days' => 14,
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],

```


Запись сообщений в конкретные каналы логирования

Иногда также бывает необходимо управлять тем, куда поступает каждый конкретный лог. Это тоже можно сделать, просто указав канал при вызове фасада Log:

```
Log::channel('slack')->info("This message will go to Slack.");
```



Расширенная настройка логирования

Если нужно настроить способ отправки логов в каждый канал или реализовать пользовательский обработчик на базе библиотеки Monolog, то об этом можно подробно прочитать в документации по логированию (<http://bit.ly/2TVgSWT>).

Полнотекстовый поиск с использованием Laravel Scout

5.3 Laravel Scout — отдельный пакет, который можно включить в состав своего приложения Laravel, чтобы снабдить свои модели Eloquent функцией полнотекстового поиска. Scout позволяет легко проиндексировать содержимое ваших моделей Eloquent и производить поиск по нему. Он поставляется с драйвером Algolia. Существуют также созданные сообществом пакеты для использования других провайдеров. Здесь я буду исходить из предположения, что вы пользуетесь Algolia.

Установка пакета Scout

В приложении, использующем Laravel 5.3 или более новую версию, сначала подгрузите данный пакет:

```
composer require laravel/scout
```



Регистрация сервис-провайдеров вручную до Laravel 5.5

Если вы используете версии до Laravel 5.5, нужно вручную зарегистрировать сервис-провайдер, добавив ссылку `Laravel\Scout\ScoutServiceProvider::class` в разделе `providers` файла `config/app.php`.

Далее настройте конфигурацию Scout следующей командой:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

Вставьте свои учетные данные Algolia в файл `config/scout.php`.

Установите Algolia SDK:

```
composer require algolia/algoliasearch-client-php
```

Пометка модели для индексирования

Импортируйте в свою модель трейт `Laravel\Scout\Searchable`. В данном примере мы будем использовать модель `Review` для рецензий на книгу.

Можно указать доступные для поиска свойства с помощью метода `toSearchableArray()`, по умолчанию он зеркалирует метод `toArray()`. Задать имя индекса модели с помощью метода `searchableAs()` (по умолчанию используется название таблицы).

Пакет Scout подписывается на события создания/удаления/обновления в помеченных вами моделях. При создании, обновлении или удалении вами любых строк Scout будет синхронизировать эти изменения с сервисом Algolia либо одновременно с внесением изменений, либо, если вы соответствующим образом настроите Scout, с помещением каждого изменения в очередь.

Поиск по вашему индексу

У пакета Scout достаточно простой синтаксис. Например, найти модель `Review`, содержащую слово `Llew`, можно так:

```
Review::search('Llew')->get();
```

Возможно модифицировать запросы, как и в случае обычных вызовов Eloquent:

```
// Получаем все записи из модели Review, дающей совпадение со словом Llew,
// с выводом до 20 записей на странице и чтением параметра запроса page,
// как в случае постраничного вывода в Eloquent
Review::search('Llew')->paginate(20);
```

```
// Получаем все записи из модели Review, дающей совпадение со словом Llew,
// и присваиваем полю account_ID значение 2
Review::search('Llew')->where('account_id', 2)->get();
```

Что возвращают такие поисковые вызовы? Коллекцию моделей Eloquent, восстановленных из вашей базы данных. Идентификаторы сохраняются в сервисе Algolia, который затем возвращает список соответствующих запросу идентификаторов. Пакет Scout извлекает для них записи БД и возвращает их в виде объектов Eloquent.

Нельзя в полной мере использовать все продвинутые возможности оператора `WHERE` языка SQL. Но вы располагаете простейшими средствами для сравнительных проверок подобно тому, как это делается в представленных здесь примерах кода.

Очереди и Scout

Пока что ваше приложение будет отправлять HTTP-запросы сервису Algolia при выполнении каждого запроса на модификацию записей базы данных. Это может быстро привести к снижению производительности вашего приложения.

Поэтому пакет Scout позволяет легко помещать все выполняемые им действия в очередь.

В файле `config/scout.php` присвойте ключу `queue` значение `true`, чтобы такие обновления индексировались в асинхронном режиме. После этого полнотекстовый индекс заработает с обеспечением окончательной согласованности. То есть обновления записей базы данных будут немедленными, а обновления поисковых индексов будут вноситься в очередь и производиться с той скоростью, какую может обеспечить используемый обработчик очередей.

Выполнение операций без индексирования

Если нужно выполнить ряд операций без запуска индексирования, оберните эти операции в метод `withoutSyncingToSearch()` модели:

```
Review::withoutSyncingToSearch(function () {  
    // Создаем несколько рецензий  
    factory(Review::class, 10)->create();  
});
```

Условное индексирование моделей

Иногда требуется индексировать записи только в случае, если они удовлетворяют определенному условию. Тут поможет вызов метода `shouldBeSearchable()` в классе модели:

```
public function shouldBeSearchable()  
{  
    return $this->isApproved();  
}
```

Запуск индексирования вручную с помощью кода

Если нужно запустить индексирование модели вручную, то воспользуйтесь кодом в вашем приложении или командной строкой.

Чтобы вручную запустить индексирование из кода, добавьте метод `searchable()` в конец любого запроса Eloquent. Он проиндексирует все найденные записи:

```
Review::all()->searchable();
```

При этом можно ограничить запрос только теми записями, которые нужно проиндексировать. Пакет Scout достаточно сообразителен, чтобы вставить новые записи и обновить старые, так что вместо этого лучше проиндексировать все содержимое таблицы базы данных модели.

Можно пристыковать метод `searchable()` к методам связей:

```
$user->reviews()->searchable();
```

Или отменить индексирование любых записей, используя такую же цепочку запроса. Нужно лишь заменить метод `searchable()` на `unsearchable()`:

```
Review::where('sucky', true)->unsearchable();
```

Запуск индексирования вручную с помощью интерфейса командной строки

Вручную запустить индексирование можно с помощью следующей команды Artisan:

```
php artisan scout:import "App\Review"
```

Эта команда извлечет и проиндексирует все модели `Review`.

Тестирование

Для проверки большинства этих элементов приложения достаточно использовать их в тестах, не прибегая к каким-либо имитациям или заглушкам. Конфигурация по умолчанию будет работать без дополнительных усилий — например, можно открыть файл `phpunit.xml` и убедиться, что там заданы подходящие для тестирования настройки драйвера сессии и кэша.

Есть еще ряд вспомогательных методов и подводных камней, о которых стоит узнать перед тестированием всех этих элементов.

Сохранение файлов

Протестировать загрузку файлов на сервер не всегда просто. После того как мы разберем описанные ниже шаги, все станет понятно.

Загрузка на сервер поддельных файлов

Сначала разберемся с созданием объекта `Illuminate\Http\UploadedFile` вручную для использования его при тестировании приложения (пример 14.18).

Пример 14.18. Создание поддельного объекта `UploadedFile` для целей тестирования

```
public function test_file_should_be_stored()
{
    Storage::fake('public');

    $file = UploadedFile::fake()->image('avatar.jpg');
```

```

$response = $this->postJson('/avatar', [
    'avatar' => $file,
]);

// Убеждаемся в том, что файл был сохранен
Storage::disk('public')->assertExists("avatars/{$file->hashName()}");

// Убеждаемся в том, что файл не существует
Storage::disk('public')->assertMissing('missing.jpg');
}

```

Таким образом, мы создали новый объект `UploadedFile`, который ссылается на наш тестовый файл. Теперь можно использовать его для проверки наших маршрутов.

Возвращение поддельных файлов

Если ваш маршрут ожидает, что есть какой-то реальный файл, то для его проверки проще всего на самом деле предоставить такой. Допустим, что профиль каждого пользователя должен иметь изображение.

Сначала настроим фабрику моделей для пользователя таким образом, чтобы она применяла `Faker` для создания копии изображения, как показано в примере 14.19.

Пример 14.19. Возвращение поддельных файлов с помощью `Faker`

```

$factory->define(User::class, function (Faker\Generator $faker) {
    return [
        'picture' => $faker->file(
            storage_path('tests'), // Исходный каталог
            storage_path('app'), // Целевой каталог
            false // Возвращаем только имя файла, а не полный путь
        ),
        'name' => $faker->name,
    ];
});

```

Метод `file()` пакета `Faker` случайным образом выбирает файл в исходном каталоге, копирует его в целевой каталог и возвращает его имя. То есть мы случайным образом выбрали файл в каталоге `storage/tests`, скопировали его в `storage/app` и присвоили его название свойству `picture` объекта `User`. После этого мы можем использовать объект `User` для тестирования маршрутов, ожидающих, что для объекта `User` будет добавлено изображение (пример 14.20).

Пример 14.20. Проверка в отношении того, отображается ли URL-адрес изображения

```

public function test_user_profile_picture_echoes_correctly()
{
    $user = factory(User::class)->create();

    $response = $this->get(route('users.show', $user->id));

    $response->assertSee($user->picture);
}

```

Во многих случаях достаточно просто сгенерировать случайную строку, даже не копируя файл. Но если ваши маршруты проверяют наличие файла или производят определенные операции над ним, то лучше воспользоваться данным способом.

Сессия

Если нужно убедиться, что в сессии присвоены определенные значения, то Laravel позволяет использовать в любом тесте описанные ниже вспомогательные методы. Все эти методы можно вызывать в тестах в объекте `Illuminate\Foundation\Testing\TestResponse`.

- ❑ `assertSessionHas($key, $value = null)`. Проверяет, содержит ли сессия значение с указанным ключом. Если передан второй параметр, то метод также проверяет, равно ли значение ключа указанному значению:

```
public function test_some_thing()
{
    // Выполнение действий, дающих на выходе объект $response ...
    $response->assertSessionHas('key', 'value');
}
```

- ❑ `assertSessionHasAll(array $bindings)`. Если передан массив пар «ключ/значение», то проверяет, присвоены ли всем указанным ключам указанные значения. Если один или несколько элементов массива представляют собой обычное значение (с использованием по умолчанию числовых ключей языка PHP), то метод просто проверяет, присутствует ли это значение в сессии:

```
$check = [
    'has',
    'hasWithThisValue' => 'thisValue',
];

$response->assertSessionHasAll($check);
```

- ❑ `assertSessionMissing($key)`. Убеждается, что сессия *не* содержит значение с указанным ключом.
- ❑ `assertSessionHasErrors($bindings = [], $format = null)`. Проверяет наличие в сессии значения с ключом `errors`. Он используется Laravel для возвращения данных об ошибках в случае неудачной валидации.

Если передается только массив ключей, то выполняется проверка на наличие ошибок с такими ключами:

```
$response = $this->post('test-route', ['failing' => 'data']);
$response->assertSessionHasErrors(['name', 'email']);
```

Можно передать значения ключей и их формат в необязательном параметре `$format`, чтобы убедиться, что возвращаемые текстовые сообщения об ошибках выглядят надлежаще:

```
$response = $this->post('test-route', ['failing' => 'data']);
$response->assertSessionHasErrors([
    'email' => '<strong>The email field is required.</strong>',
], '<strong>:message</strong>');
```

Кэш

Тестирование использующих кэш элементов приложения не представляет ничего сложного:

```
Cache::put('key', 'value', 900);

$this->assertEquals('value', Cache::get('key'));
```

По умолчанию Laravel использует в среде тестирования драйвер кэша `array`, который сохраняет кэшируемые значения в памяти.

Cookie-файлы

Что, если нужно установить cookie-файлы перед проверкой маршрута в тестах приложений? Можно вручную передать cookie-файл в качестве параметра методу `call()`. Подробные сведения о методе `call()` были приведены в главе 12.



Отмена шифрования cookie-файла на время тестирования

Cookie-файлы не будут работать в тестах, если вы не отмените их шифрование с помощью предлагаемого Laravel middleware для шифрования cookie-файлов. Для этого временно отмените действие middleware `EncryptCookies` для этих cookie-файлов с помощью метода `disableFor()`:

```
use Illuminate\Cookie\Middleware\EncryptCookies;
...

$this->app->resolving(
    EncryptCookies::class,
    function ($object) {
        $object->disableFor('cookie-name');
    }
);

// ... выполнение теста
```

Это означает, что вы можете выполнить установку cookie-файлов и затем проверить их наличие приблизительно так, как в примере 14.21.

Пример 14.21. Выполнение модульных тестов в отношении cookie-файлов

```
public function test_cookie()
{
    $this->app->resolving(EncryptCookies::class, function ($object) {
        $object->disableFor('my-cookie');
    });

    $response = $this->call(
        'get',
        'route-echoing-my-cookie-value',
        [],
        ['my-cookie' => 'baz']
    );
    $response->assertSee('baz');
}
```

Если нужно проверить, установлен ли в ответе некоторый cookie-файл, проверьте его наличие с помощью метода `assertCookie()`:

```
$response = $this->get('cookie-setting-route');
$response->assertCookie('cookie-name');
```

Методом `assertPlainCookie()` можно убедиться в наличии незашифрованного cookie-файла.



Отличия в наименовании методов тестирования в версиях, предшествующих Laravel 5.4

В проектах, использующих версии до Laravel 5.4, метод `assertCookie()` следует заменить методом `seeCookie()`, а `assertPlainCookie()` — `seePlainCookie()`.

Логирование

Самый простой способ убедиться, что был записан определенный лог, — проверить утверждения в отношении фасада `Log` (см. подробности в подразделе «Подделка других фасадов» на с. 334). Как это можно сделать, показано в примере 14.22.

Пример 14.22. Проверка утверждений в отношении фасада `Log`

```
// Тестовый файл
public function test_new_accounts_generate_log_entries()
{
    Log::shouldReceive('info')
        ->once()
        ->with('New account created!');
```



```
// Создание новой учетной записи
$this->post(route('accounts.store'), ['email' => 'matt@mattstauffer.com']);
}

// AccountsController
public function store()
{
    // Создание учетной записи

    Log::info('New account created!');
}
```

Можно использовать пакет под названием Log Fake (<http://bit.ly/2JDI4vd>), который расширяет показанные здесь возможности тестирования фасада, позволяя проверять более тонко настроенные утверждения в отношении ваших логов.

Scout

Когда требуется проверить код, использующий данные пакета Scout, обычно нужно сделать так, чтобы в тестах не выполнялось индексирование или получение данных от пакета Scout. Для этого добавьте в файл `phpunit.xml` переменную среды, отключающую подключение Scout к сервису Algolia:

```
<env name="SCOUT_DRIVER" value="null"/>
```

Резюме

Laravel предоставляет простые интерфейсы для множества распространенных операций сохранения — доступа к файловой системе, сессиям, cookie-файлам, кэшу и поиску. Каждый из этих API остается неизменным вне зависимости от используемого провайдера. Laravel позволяет различным «драйверам» обслуживать один и тот же публичный интерфейс. Так легко переключаться между разными провайдерами при смене среды или изменении потребностей приложения.

Уведомление пользователей приложения посредством электронной почты, сервиса Slack, СМС-сервиса или другой системы уведомлений — весьма распространенное, но сложно реализуемое требование. Компоненты Laravel для работы с почтой и уведомлениями предлагают унифицированные API, которые позволяют в значительной мере абстрагироваться от того, какой именно провайдер вы выбрали. Подобно тому как это делалось в главе 14, нужно лишь один раз написать свой код и указать на уровне конфигурации, какой провайдер будет использоваться для отправки электронной почты или уведомлений.

Почта

Функциональность электронной почты Laravel — вспомогательный слой поверх библиотеки Swift Mailer по адресу <http://swiftmailer.org/>, и «из коробки» Laravel предлагает драйверы для Mailgun, Mandrill, Sparkpost, SES, SMTP, PHP Mail и Sendmail.

Для всех облачных сервисов следует указать аутентификационные данные в файле `config/services.php`. Однако при ближайшем рассмотрении видно, что в этом файле и в `config/mail.php` уже есть ключи, позволяющие настроить функциональность электронной почты вашего приложения в файле `.env`, используя переменные `MAIL_DRIVER` и `MAILGUN_SECRET`.



Зависимости драйверов облачных API

Если вы применяете драйверы каких-либо облачных API, то нужно подгрузить пакет Guzzle с помощью утилиты Composer. Воспользуйтесь следующей командой:

```
composer require guzzlehttp/guzzle
```

Если у вас драйвер сервиса SES, выполните команду:

```
composer require aws/aws-sdk-php:~3.0
```

Классическая электронная почта

В Laravel есть два различных синтаксиса электронной почты: классический и синтаксис отправлений (mailable). С версии 5.3 рекомендуется синтаксис отправлений, поэтому в книге мы сосредоточимся на таком варианте. Однако для тех читателей, которые работают с версией 5.2 или предыдущими, ниже представлен пример использования классического синтаксиса (пример 15.1).

Пример 15.1. Простейший пример классического синтаксиса почты

```
Mail::send(
    'emails.assignment-created',
    ['trainer' => $trainer, 'trainee' => $trainee],
    function ($m) use ($trainer, $trainee) {
        $m->from($trainer->email, $trainer->name);
        $m->to($trainee->email, $trainee->name)->subject('A New Assignment!');
    }
);
```

В качестве первого параметра методу `Mail::send()` передается имя представления. Здесь подразумевается, что за названием `emails.assignment-created` скрывается файл `resources/views/emails/assignment-created.blade.php` или `resources/views/emails/assignment-created.php`.

Второй — массив данных, которые нужно передать в представление.

Третьим параметром передается замыкание, которое определяет, как и куда следует отправить электронное письмо — адреса отправителя, получателя, получателя копии и скрытой копии, тема письма и все необходимые метаданные. Любые используемые в замыкании переменные необходимо объявить с помощью ключевого слова `use`. Обратите внимание, что замыканию передается единственный параметр, в данном случае с именем `$m`. Это объект сообщения.

Подробные сведения о классическом синтаксисе электронной почты можно найти в старой документации по адресу <http://bit.ly/2utCAZA>.

Простейший способ использования отправлений

53 В Laravel 5.3 был введен новый синтаксис электронной почты, основанный на использовании так называемых отправлений (mailable). Он действует аналогично классическому. Но вместо того, чтобы определять свои почтовые сообщения в замыкании, вы создаете отдельные РНР-классы для представления каждого письма.

Такое отправление можно сгенерировать с помощью команды `Artisan make:mail`:

```
php artisan make:mail AssignmentCreated
```

Этот класс показан в примере 15.2.

Пример 15.2. Автоматически генерируемый РНР-класс отправления

```
<?php
```

```
namespace App\Mail;
```

```
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class AssignmentCreated extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Создание нового экземпляра сообщения
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Построение сообщения
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('view.name');
    }
}
```

Данный класс по своей структуре мало отличается от класса `Job`. Он даже импортирует трейт `Queueable` для постановки писем в очередь и трейт `SerializesModels` для корректной сериализации моделей Eloquent, передаваемых вами конструктору.

Как же все это работает? В методе `build()` класса отправления задаются используемое представление, тема и остальные параметры письма, которые необходимо настроить, *за исключением адреса получателя*. Необходимые данные передаются через конструктор, а все публичные свойства класса отправления доступны шаблону.

В примере 15.3 показано, как модифицировать автоматически генерируемый класс отправления для нашего примера с письмом о назначении.

Пример 15.3. Пример класса отправления

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class AssignmentCreated extends Mailable
{
```

```

use Queueable, SerializesModels;

public $trainer;
public $trainee;

public function __construct($trainer, $trainee)
{
    $this->trainer = $trainer;
    $this->trainee = $trainee;
}

public function build()
{
    return $this->subject('New assignment from ' . $this->trainer->name)
        ->view('emails.assignment-created');
}
}

```

Как можно отослать отправление, см. в примере 15.4.

Пример 15.4. Несколько способов отсылки отправлений

```

// Простая отправка
Mail::to($user)->send(new AssignmentCreated($trainer, $trainee));

// С направлением копии/скрытой копии/и т.д.
Mail::to($user1)
    ->cc($user2)
    ->bcc($user3)
    ->send(new AssignmentCreated($trainer, $trainee));

// С использованием коллекций
Mail::to('me@app.com')
    ->bcc(User::all())
    ->send(new AssignmentCreated($trainer, $trainee))

```

Шаблоны писем

Шаблоны писем ничем не отличаются от любых других. Они могут расширять иные шаблоны, использовать разделы, выполнять синтаксический разбор переменных, содержать условные или циклические директивы и делать все, что доступно в представлении Blade.

В примере 15.5 показан возможный вариант шаблона `emails.assignment-created` для примера 15.3.

Пример 15.5. Возможный вариант шаблона электронного письма о назначении

```

<!-- resources/views/emails/assignment-created.blade.php -->
<p>Hey {{ $trainee->name }}!</p>

<p>You have received a new training assignment from <b>{{ $trainer->name }}</b>.
Check out your <a href="{{ route('training-dashboard') }}">training
dashboard</a> now!</p>

```

В примере 15.3 переменные `$trainer` и `$trainee` — публичные свойства класса отправления, что делает их доступными для шаблона.

При желании можно явно указать, какие переменные следует передавать шаблону, пристыковав метод `with()` к вызову метода `build()`, как в примере 15.6.

Пример 15.6. Определение переменных шаблона

```
public function build()
{
    return $this->subject('You have a new assignment!')
        ->view('emails.assignment')
        ->with(['assignment' => $this->event->name]);
}
```



HTML-письма и простые текстовые письма

В примерах выше в стеке вызовов метода `build()` использовался метод `view()`. В таком случае указанный нами шаблон будет возвращать HTML-версию письма. Если нужна простая текстовая версия, примените метод `text()`, который определяет простое текстовое представление:

```
public function build()
{
    return $this->view('emails.reminder')
        ->text('emails.reminder_plain');
}
```

Методы, доступные в `build()`

Ниже перечислены еще несколько методов, которые можно использовать в методе `build()` класса отправления для настройки своего сообщения.

- ❑ `from($address, $name = null)`. Задаёт адрес и имя отправителя.
- ❑ `subject($subject)`. Задаёт тему письма.
- ❑ `attach($file, array $options = [])`. Прикрепляет файл. Допустимые параметры — `mime` для MIME-типа и `as` для отображаемого имени.
- ❑ `attachData($data, $name, array $options = [])`. Прикрепляет файл из необработанной строки. Принимает те же параметры, что и метод `attach()`.
- ❑ `attachFromStorage($path, $name = null, array $options = [])`. Прикрепляет файл, сохранённый на любом из дисков файловой системы.
- ❑ `priority($level = n)`. Задаёт приоритет электронного письма в диапазоне от 1 до 5.

Можно вручную модифицировать базовое сообщение библиотеки Swift Mailer, воспользовавшись методом `withSwiftMessage()`, как показано в примере 15.7.

Пример 15.7. Модификация базового объекта SwiftMessage

```
public function build()
{
    return $this->subject('Howdy!')
        ->withSwiftMessage(function ($swift) {
            $swift->setReplyTo('noreply@email.com');
        })
        ->view('emails.howdy');
}
```

Прикрепленные файлы и встроенные изображения

Пример 15.8 демонстрирует три возможных варианта прикрепления к электронному письму файлов или необработанных данных.

Пример 15.8. Прикрепление файлов или данных к отправлениям

```
// Прикрепляем файл, используя локальное имя файла
public function build()
{
    return $this->subject('Your whitepaper download')
        ->attach(storage_path('pdfs/whitepaper.pdf'), [
            'mime' => 'application/pdf', // Опционально
            'as' => 'whitepaper-barasa.pdf', // Опционально
        ])
        ->view('emails.whitepaper');
}

// Прикрепляем файл, передавая необработанные данные
public function build()
{
    return $this->subject('Your whitepaper download')
        ->attachData(
            file_get_contents(storage_path('pdfs/whitepaper.pdf')),
            'whitepaper-barasa.pdf',
            [
                'mime' => 'application/pdf', // Опционально
            ]
        )
        ->view('emails.whitepaper');
}

// Прикрепляем файл, сохраненный на одном из дисков файловой системы
public function build()
{
    return $this->subject('Your whitepaper download')
        ->view('emails.whitepaper')
        ->attachFromStorage('/pdfs/whitepaper.pdf');
}
```

Как встраивать непосредственно в электронное письмо изображения, показано в примере 15.9.

Пример 15.9. Встраивание изображений

```
<!-- emails/image.blade.php -->
Here is an image:
```

```

```

Or, the same image embedding the data:

```

```

Markdown-отправления

Markdown-отправления позволяют писать содержание своих электронных писем в виде Markdown-разметки, а затем преобразовывать их в полноценные HTML-письма (или простые текстовые) с помощью встроенных, легко адаптируемых HTML-шаблонов Laravel. Можно настроить эти шаблоны на создание пользовательского шаблона электронного письма, упрощающего создание содержимого для ваших разработчиков и других пользователей.

Для начала следует выполнить команду `Artisan make:mail` с флагом `markdown`:

```
php artisan make:mail AssignmentCreated --markdown=emails.assignment-created
```

Как может выглядеть генерируемый этой командой файл письма, см. в примере 15.10.

Пример 15.10. Сгенерированное Markdown-отправление

```
class AssignmentCreated extends Mailable
{
    // ...

    public function build()
    {
        return $this->markdown('emails.assignment-created');
    }
}
```

Данный файл почти аналогичен обычному файлу отправления в Laravel. Основное отличие — вместо метода `view()` здесь вызывается метод `markdown()`. Следует иметь в виду, что указываемый здесь шаблон должен представлять собой шаблон Markdown, а не обычный шаблон Blade.

Если обычный шаблон электронного письма генерирует полноценное HTML-письмо, используя для этого включения и наследования, как любой другой файл Blade, то шаблоны Markdown просто передают содержание письма нескольким заранее определенным компонентам. В Laravel вложение компонентов уровня фреймворка и пакета часто осуществляется с помощью имен вида `mail::button`. Соответственно, основное содержание Markdown-письма должно быть передано

компоненту с названием `mail::message`. Пример 15.11 демонстрирует простейший Markdown-шаблон письма.

Пример 15.11. Простейшее Markdown-письмо о назначении

```
{{{-- resources/views/emails/assignment-created.blade.php --}}
@component('mail::message')
# Hey {{ $trainee->name }}!

You have received a new training assignment from **{{ $trainer->name }}**

@component('mail::button', ['url' => route('training-dashboard')])
View Your Assignment
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

Как видно из примера 15.11, помимо родительского компонента `mail::message`, которому передается тело электронного письма, также предоставляется ряд более мелких, которые легко вкраплять в свои электронные письма. Мы использовали здесь `mail::button`, принимающий тело сообщения (`View Your Assignment` — «Просмотр назначения»), и массив параметров, передаваемый в качестве второго параметра директивы `@component`.

Компоненты Markdown

Имеется три типа компонентов.

- ❑ **button.** Генерирует центрированную ссылку-кнопку. Компонент `button` принимает обязательный атрибут `url` и необязательный `color`, в качестве которого можно передать значения `primary`, `success` и `error`.
- ❑ **panel.** Отображает предоставленный текст, используя чуть более светлый фон по сравнению с остальной частью сообщения.
- ❑ **table.** Преобразует переданное содержимое табличным синтаксисом языка Markdown.



Настройка компонентов

Эти компоненты Markdown встроены в ядро Laravel. Если нужно настроить способ их работы, можно опубликовать и отредактировать их файлы:

```
php artisan vendor:publish --tag=laravel-mail
```

В документации Laravel по адресу <http://bit.ly/2UUBuRf> содержатся более подробные сведения о настройке этих файлов и их темах.

Визуализация отправлений в браузере

5.5 При разработке электронных писем для приложений полезен предварительный просмотр их внешнего вида. Для этого подходит сервис Mailtrap, но иногда лучше отображать письма непосредственно в браузере и сразу видеть результат производимых изменений.

В примере 15.12 показано, как в приложение добавить маршрут для визуализации определенного отправления.

Пример 15.12. Маршрут для визуализации отправления

```
Route::get('preview-assignment-created-mailable', function () {
    $trainer = Trainer::first();
    $trainee = Trainee::first();

    return new \App\Mail\AssignmentCreated($trainer, $trainee);
});
```

Очереди

Отправка электронных писем требует определенных затрат времени, которые могут снизить производительность вашего приложения. Чтобы этого не происходило, письма помещают в фоновую очередь. Это настолько общепринятая практика, что в Laravel включен набор инструментов, позволяющих легко помещать письма в очередь без необходимости писать задания очереди для каждого отдельного письма.

❑ `queue()`. Для помещения объекта отправления в очередь вместо его немедленной отправки передайте его методу `Mail::queue()`, а не `Mail::send()`:

```
Mail::queue(new AssignmentCreated($trainer, $trainee));
```

❑ `later()`. Метод `Mail::later()` аналогичен `Mail::queue()`, но позволяет добавить задержку, задав величину задержки в минутах либо передав объект `DateTime` или `Carbon`, — указывает, когда нужно извлечь из очереди и отправить электронное письмо:

```
$when = now()->addMinutes(30);
Mail::later($when, new AssignmentCreated($trainer, $trainee));
```



Настройка очередей

Эти методы будут работать лишь при правильной настройке своих очередей. В главе 16 подробно рассказано о принципе действия очередей и о том, как можно заставить их работать в своем приложении.

В случае методов `queue()` и `later()` можно указать, какую именно очередь и подключение к очереди следует использовать для вашего письма, вызвав методы `onConnection()` и `onQueue()` в объекте отправления:

```
$message = (new AssignmentCreated($trainer, $trainee))
    ->onConnection('sqs')
    ->onQueue('emails');
```

```
Mail::to($user)->queue($message);
```

Если нужно, чтобы определенное отправление всегда помещалось в очередь, реализуйте в его классе интерфейс `Illuminate\Contracts\Queue\ShouldQueue`.

Локальная разработка

Сказанное выше хорошо подходит для отправки почты в среде эксплуатации. Но как это можно протестировать? Для этого подходит один из трех основных инструментов: драйвер логирования `log` фреймворка Laravel, SaaS-приложение Mailtrap и конфигурационный параметр «универсального получателя».

Драйвер логирования

Laravel предоставляет драйвер логирования `log`, который регистрирует каждую попытку отправки электронных писем в локальном файле `laravel.log` (по умолчанию находится в папке `storage/logs`).

Чтобы воспользоваться данным драйвером, необходимо отредактировать содержимое файла `.env`, присвоив ключу `MAIL_DRIVER` значение `log`. После этого при отправке приложением электронного письма в файле `storage/logs/laravel.log` будет появляться следующая запись:

```
Message-ID: <04ee2e97289c68f0c9191f4b04fc0de1@localhost>
Date: Tue, 17 May 2016 02:52:46 +0000
Subject: Welcome to our app!
From: Matt Stauffer <matt@mattstauffer.com>
To: freja@jensen.no
MIME-Version: 1.0
Content-Type: text/html; charset=utf-8
Content-Transfer-Encoding: quoted-printable
```

```
Welcome to our app!
```

С версии 5.7 можно дополнительно указать отдельный канал записи для логирования отправки писем. Нужно модифицировать файл `config/mail.php` или присвоить имя любого имеющегося канала логирования переменной `MAIL_LOG_CHANNEL` в файле `.env`.

Mailtrap.io

Mailtrap (<https://mailtrap.io>) — это сервис для перехвата и проверки электронных писем в среде разработки. Вы отправляете электронные письма серверам сервиса Mailtrap, используя протокол SMTP, но, вне зависимости от адреса электронной почты в поле `to` (получатель), Mailtrap не отправляет эти письма указанному получателю, а перехватывает, позволяя проверить их веб-клиентом электронной почты.

Чтобы воспользоваться сервисом Mailtrap, оформите бесплатную учетную запись. После откройте демонстрационный почтовый ящик на основной информационной панели. Скопируйте оттуда имя пользователя и пароль для SMTP-соединения.

Затем отредактируйте файл `.env` вашего приложения, указав следующие значения в разделе `mail`:

```
MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=your_username_from_mailtrap_here
MAIL_PASSWORD=your_password_from_mailtrap_here
MAIL_ENCRYPTION=null
```

Теперь любое отправляемое вашим приложением письмо будет появляться в почтовом ящике, предоставленном сервисом Mailtrap.

Универсальный получатель

Можно проверять электронные письма в предпочитаемом вами клиенте, переопределив поле `to` каждого сообщения конфигурационным параметром «универсального получателя». Для этого добавьте в файле `config/mail.php` ключ `to` примерно следующего вида:

```
'to' => [
    'address' => 'matt@mattstauffer.com',
    'name' => 'Matt Testing My Application'
],
```

Для этого способа потребуется настроить реальный драйвер электронной почты, использующий сервис Mailgun или Sendmail.

Уведомления

Большинство отправляемых веб-приложением электронных писем в действительности служат для уведомления пользователей, что произошло или должно произойти определенное действие. По мере того как предпочтения пользователей в отношении способа коммуникации становятся разнообразнее, нам нужно все

больше разнородных пакетов для коммуникации сервиса Slack, СМС-сервиса и других средств коммуникации.

В Laravel 5.3 введена новая концепция с подходящим названием *уведомления*. Подобно отправлению, это РНР-класс для представления одного из отсылаемых пользователям сообщений. В качестве примера предположим, что наше приложение для тренеров и спортсменов должно уведомить пользователей о доступности нового вида тренировки.

Каждый такой класс — вся информация, необходимая для отправки пользователям уведомлений *посредством одного или нескольких каналов*. Отдельное уведомление может отправлять электронное письмо или СМС-сообщение с помощью сервиса Nexmo, направлять проверочный сигнал веб-сокетами, добавлять запись в базу данных, отправлять сообщение в Slack-канал и т. д.

Создадим уведомление:

```
php artisan make:notification WorkoutAvailable
```

Что мы получим в результате, см. в примере 15.13.

Пример 15.13. Автоматически генерируемый класс уведомления

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class WorkoutAvailable extends Notification
{
    use Queueable;

    /**
     * Создание нового экземпляра уведомления
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Получение каналов доставки уведомления
     *
     * @param mixed $notifiable
     * @return array
     */
}
```

```
public function via($notifiable)
{
    return ['mail'];
}

/**
 * Получение письма, представляющего уведомление
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->line('The introduction to the notification.')
        ->action('Notification Action', url('/'))
        ->line('Thank you for using our application!');
}

/**
 * Получение массива, представляющего уведомление
 *
 * @param mixed $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        //
    ];
}
}
```

Пример показывает следующее. Во-первых, необходимые данные передаются в конструктор. Во-вторых, используя метод `via()`, можно указать, какие каналы уведомлений нужны для определенного пользователя. Переменная `$notifiable` представляет уведомляемые приложением сущности. Обычно это пользователь, но бывают и исключения из правил. В-третьих, для каждого канала уведомлений предусмотрен отдельный метод, позволяющий определить конкретный способ отправки уведомлений этим каналом.



Когда переменная `$notifiable` не представляет пользователя

Обычно уведомления направляются пользователю. Не исключена ситуация, когда требуется уведомить кого-то другого. К примеру, у приложения несколько типов пользователей — вам нужно уведомлять и тренеров, и спортсменов. Но иногда нужно уведомить группу пользователей, компанию или сервер.

Таким образом, модифицируем этот класс, чтобы получить уведомление о доступности нового вида тренировки `WorkoutAvailable`. Такой класс показан в примере 15.14.

Пример 15.14. Наш класс уведомления WorkoutAvailable

```
...
class WorkoutAvailable extends Notification
{
    use Queueable;

    public $workout;

    public function __construct($workout)
    {
        $this->workout = $workout;
    }

    public function via($notifiable)
    {
        // Этого метода нет в классе User... чуть позже мы это исправим
        return $notifiable->preferredNotificationChannels();
    }

    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->line('You have a new workout available!')
            ->action('Check it out now', route('workout.show', [$this->workout]))
            ->line('Thank you for training with us!');
    }

    public function toArray($notifiable)
    {
        return [];
    }
}
```

Определение метода via() для уведомляемых объектов

Как видно из примера 15.14, мы должны определенным образом указать, какие каналы уведомлений следует использовать для каждого уведомления и уведомляемого объекта.

Можно отправлять все в виде электронного письма или СМС (пример 15.15).

Пример 15.15. Простейшая возможная реализация метода via()

```
public function via($notifiable)
{
    return 'nexmo';
}
```

Можно позволить каждому пользователю указать предпочитаемый способ коммуникации и сохранить его выбор непосредственно в объекте User (пример 15.16).

Пример 15.16. Индивидуальная настройка метода `via()` для каждого пользователя

```
public function via($notifiable)
{
    return $notifiable->preferred_notification_channel;
}
```

Еще одна возможность — создать в каждом уведомляемом объекте метод, содержащий некоторую сложную логику уведомления, как предполагалось в примере 15.14. Скажем, можно использовать определенный ряд каналов для уведомления пользователя в рабочее время, и другой ряд — для уведомления его в вечернее время. Важный момент — `via()` представляет собой метод PHP-класса и может содержать даже очень сложную логику.

Отправка уведомлений

Отправить уведомление можно двумя способами — с помощью фасада `Notification` или путем добавления трейта `Notifiable` в класс `Eloquent` (обычно класс `User`).

Отправка уведомлений с использованием трейта `Notifiable`

Любая модель, которая импортирует трейт `Laravel\Notifications\Notifiable` (класс `App\User` делает по умолчанию), поддерживает метод `notify()`, которому можно передать уведомление, выглядящее примерно так, как в примере 15.17.

Пример 15.17. Отправка уведомления с использованием трейта `Notifiable`

```
use App\Notifications\WorkoutAvailable;
...
$user->notify(new WorkoutAvailable($workout));
```

Отправка уведомлений с использованием фасада `Notification`

Использовать фасад `Notification` сложнее, поскольку нужно передать уведомление и уведомляемый объект. Но это дает возможность отправить сразу несколько уведомляемых объектов, как показано в примере 15.18.

Пример 15.18. Отправка уведомлений с использованием фасада `Notification`

```
use App\Notifications\WorkoutAvailable;
...
Notification::send($users, new WorkoutAvailable($workout));
```

Помещение уведомлений в очередь

Большинство драйверов уведомлений отправляет уведомления путем отправки HTTP-запросов, что может привести к замедлению пользовательского интерфейса. Чтобы этого не происходило, следует помещать уведомления в очередь. Поскольку

все уведомления по умолчанию импортируют трейт `Queueable`, вам остается лишь добавить в свое уведомление строку `implements ShouldQueue`, и Laravel сразу же внесет его в очередь.

Нужно убедиться, что должным образом настроены параметры очереди и работает обработчик очередей.

Если требуется, чтобы уведомление доставлялось с задержкой, вызовите метод `delay()` в уведомлении:

```
$delayUntil = now()->addMinutes(15);

$user->notify((new WorkoutAvailable($workout))->delay($delayUntil));
```

Предлагаемые по умолчанию типы уведомлений

По умолчанию Laravel предлагает драйверы для отправки уведомлений с использованием электронной почты, базы данных, возможностей трансляции, сервисов СМС-уведомлений Nexmo и Slack. Кратко рассмотрим каждый из этих каналов. Подробную информацию смотрите в документации по адресу <http://bit.ly/2JC2TqQ>.

Можно легко создать собственные драйверы уведомлений, и многие разработчики уже успели это сделать. Созданные ими драйверы есть на сайте Laravel Notification Channels по адресу <http://bit.ly/2YmpHOF>.

Почтовые уведомления

Посмотрим на структуру электронного письма из рассмотренного нами ранее примера 15.14:

```
public function toMail($notifiable)
{
    return (new MailMessage)
        ->line('You have a new workout available!')
        ->action('Check it out now', route('workouts.show', [$this->workout]))
        ->line('Thank you for training with us!');
}
```

Результат показан на рис. 15.1. Система почтовых уведомлений использует в качестве заголовка письма название вашего приложения. Его можно отредактировать в файле `config/app.php` в ключе `name`.

Это электронное письмо автоматически отправляется свойству `email` уведомляемого объекта, но можно изменить это поведение, добавив в класс уведомляемого объекта метод `routeNotificationForMail()`, возвращающий адрес электронной почты, на который нужно отправлять почтовые уведомления.

Тема электронного письма задается путем синтаксического разбора имени класса уведомления и преобразования его в слова. Так, уведомлению `WorkoutAvailable` по умолчанию будет присвоена тема `Workout Available` (Доступна тренировка). Это изменяется дополнительным вызовом метода `subject()` в объекте `MailMessage` внутри метода `toMail()`.

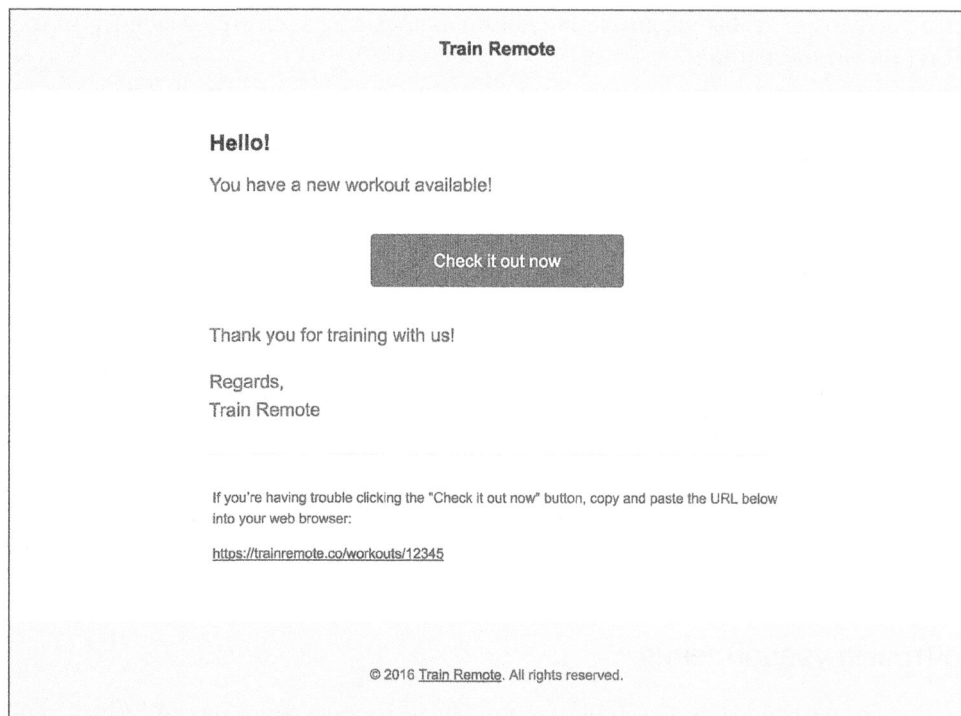


Рис. 15.1. Электронное письмо, отправленное с помощью стандартного шаблона уведомлений

Шаблоны модифицируются. Для этого опубликуйте их и отредактируйте нужным вам образом:

```
php artisan vendor:publish --tag=laravel-notifications
```

Почтовые Markdown-уведомления

Если нужны Markdown-письма (см. подраздел «Markdown-отправления» на с. 424), то вызывайте в уведомлениях тот же метод `markdown()`, как показано в примере 15.19:

Пример 15.19. Использование метода `markdown()` для отправки уведомлений

```
public function toMail($notifiable)
{
```

```
return (new MailMessage)
    ->subject('Workout Available')
    ->markdown('emails.workout-available', ['workout' => $this->workout]);
}
```

Стиль шаблона по умолчанию заменяется стилем сообщения об ошибке, у которого немного другой текст и основной цвет кнопки меняется на красный. Для этого добавьте вызов метода `error()` в цепочку вызовов объекта `MailMessage` внутри метода `toMail()`.

База данных для уведомлений

Отправлять уведомления в таблицу базы данных можно каналом уведомлений `database`. Для этого создайте таблицу с помощью команды `php artisan notifications:table`. Затем напишите в своем уведомлении метод `toDatabase()`, возвращающий массив. Эти данные будут кодироваться в формат JSON и сохраняться в столбце `data` таблицы БД.

Трейт `Notifiable` снабжает любую модель, в которую импортируется, связью `notifications`. Это дает доступ к записям в таблице уведомлений. То есть вы можете работать с уведомлениями в базе данных так, как в примере 15.20.

Пример 15.20. Перебор уведомлений в базе данных пользователя

```
User::first()->notifications->each(function ($notification) {
    // Выполнение определенных действий
});
```

Канал уведомлений `database` позволяет различать прочитанные и непрочитанные уведомления. Так, можно ограничиться только непрочитанными уведомлениями, как показано в примере 15.21.

Пример 15.21. Перебор непрочитанных уведомлений в базе данных пользователя

```
User::first()->unreadNotifications->each(function ($notification) {
    // Выполнение определенных действий
});
```

Можете пометить одно/все уведомления как прочитанные (пример 15.22).

Пример 15.22. Пометка уведомлений в базе данных как прочитанных

```
// Отдельное уведомление
User::first()->unreadNotifications->each(function ($notification) {
    if ($condition) {
        $notification->markAsRead();
    }
});

// Все уведомления
User::first()->unreadNotifications->markAsRead();
```

Трансляция уведомлений

Канал `broadcast` отправляет уведомления, используя предлагаемые Laravel возможности трансляции событий, реализуемые с помощью веб-сокетов. Подробнее о них будет рассказано в разделе «Трансляция событий посредством веб-сокетов и Laravel Echo» на с. 454.

Создайте в своем уведомлении метод `toBroadcast()`, возвращающий массив данных. Если ваше приложение должным образом настроено для трансляции событий, эти данные начнут транслироваться через приватный канал с именем `notifiable.id`. Здесь `id` — это идентификатор уведомляемого объекта, а `notifiable` — полное имя класса уведомляемого объекта, в котором знаки косой черты заменены на точки — так, например, объекту `App\User` с идентификатором 1 будет назначен приватный канал `App.User.1`.

СМС-уведомления

СМС-уведомления отправляются сервисом Nexmo (<https://www.nexmo.com>), поэтому для использования этого канала следует оформить учетную запись сервиса Nexmo и выполнить приведенные в документации инструкции (<http://bit.ly/2JC2TqQ>). Как и в случае других каналов, нужно создать специальный метод `toNexmo()` и настроить в нем СМС нужным вам образом.



Начиная с версии 5.8, выделен отдельный пакет СМС-уведомлений

 С версии 5.8 канал СМС-уведомлений выделен в дополнительно загружаемый пакет. Поэтому, если нужно использовать СМС-уведомления на базе сервиса Nexmo, просто запросите этот пакет утилитой Composer:

```
composer require laravel/nexmo-notification-channel
```

Slack-уведомления

Канал уведомлений `slack` позволяет настраивать внешний вид своих уведомлений и даже прикреплять к ним файлы. Потребуется создать специальный метод `toSlack()` и настроить сообщения.



Начиная с версии 5.8, выделен отдельный пакет Slack-уведомлений

 С версии 5.8 канал Slack-уведомлений выделен в дополнительно загружаемый пакет. Если нужны Slack-уведомления, просто запросите этот пакет с помощью утилиты Composer.

```
composer require laravel/slack-notification-channel
```

Другие виды уведомлений

Вам нужно отправлять уведомления иными каналами, помимо тех, что по умолчанию? Воспользуйтесь результатом усилий сообщества разработчиков по обеспечению широкого разнообразия каналов уведомлений — созданные им драйверы есть на сайте [Laravel Notification Channels](http://bit.ly/2YmpHOF) (<http://bit.ly/2YmpHOF>).

Тестирование

Теперь посмотрим, как проходит тестирование электронной почты и уведомлений.

Электронная почта

Laravel предлагает два способа тестирования электронной почты. Если вы используете традиционный синтаксис электронной почты (не рекомендуется использовать с версии 5.3), то подходит инструмент под названием MailThief (<http://bit.ly/2CCJ4K6>), созданный Адамом Уотаном для компании Tighten. Загрузив его в свое приложение с помощью утилиты Composer, можно вызвать в тесте метод `MailThief::hijack()`, чтобы заставить MailThief перехватывать все вызовы фасада `Mail` или любых других классов для работы с почтой.

После этого MailThief позволяет проверить утверждения в отношении адреса отправителя, получателя, получателя копии и скрытой копии, а также содержания и прикрепленных файлов. Для подробных сведений посетите сайт сервиса GitHub или загрузите MailThief в приложение:

```
composer require tightenco/mailthief -dev
```

Если вы используете отправления, то для проверки утверждений в отношении отправляемой почты подходит показанный ниже простой синтаксис (пример 15.23).

Пример 15.23. Проверка утверждений в отношении отправлений

```
public function test_signup_triggers_welcome_email()
{
    Mail::fake();

    Mail::assertSent(WelcomeEmail::class, function ($mail) {
        return $mail->subject == 'Welcome!';
    });

    // Можно также использовать метод assertSentTo() для явной проверки получателей,
    // а также метод assertNotSent(), чтобы убедиться, что определенное письмо
    // не было отправлено.
}
```

Уведомления

Laravel предлагает встроенный набор утверждений для тестирования ваших уведомлений. Их применение демонстрирует пример 15.24.

Пример 15.24. Проверка на предмет того, были ли отправлены уведомления

```
public function test_new_signups_triggers_admin_notification()
{
    Notification::fake();

    Notification::assertSentTo($user, NewUsersSignedup::class,
        function ($notification, $channels) {
            return $notification->user->email == 'user-who-signed-up@gmail.com'
                && $channels == ['mail'];
        });

    // Убеждаемся в том, что электронное письмо было отправлено
    //указанному пользователю
    Notification::assertSentTo(
        [$user],
        NewUsersSignedup::class
    );

    // Также можно использовать метод assertNotSentTo()
    Notification::assertNotSentTo(
        [$userDidntSignUp], NewUsersSignedup::class
    );
}
```

Резюме

Компоненты Laravel для работы с почтой и уведомлениями предлагают простые унифицированные интерфейсы для широкого набора систем обмена сообщениями. Почтовая система Laravel использует отправляемые (mailable) — PHP-классы, которые представляют электронные письма, чтобы предоставить унифицированный синтаксис различным драйверам электронной почты. Системой уведомлений легко создать отдельное уведомление, которое может быть доставлено посредством самых разных каналов доставки — от электронных писем и СМС до настоящих почтовых открыток.

16

Очереди, задания, события, трансляция и планировщик

Мы рассмотрели несколько наиболее распространенных структур веб-приложения, таких как базы данных, электронная почта, файловые системы и т. д. Они часто используются в большинстве приложений и фреймворков.

Laravel поддерживает еще несколько менее распространенных архитектурных шаблонов и структур приложения. В этой главе мы рассмотрим инструменты фреймворка Laravel для реализации очередей, заданий, событий и публикации событий посредством веб-сокетов. Поговорим о планировщике Laravel, который позволит вам забыть о редактируемых вручную графиках cron как о пережитке прошлого.

Очереди

Чтобы понять, что представляет собой очередь, достаточно вспомнить, как вы становитесь в очередь в банке. При наличии нескольких очередей за раз обслуживают по одной персоне из каждой. Все люди в итоге достигают начала очереди и обслуживаются. В некоторых банках строго соблюдается принцип «первым прибыл — первым обслужен», в то время как в других нет абсолютной гарантии того, что кто-то не влезет без очереди. То есть после добавления в очередь кто-то из клиентов может быть преждевременно из нее удален или успешно обслужен, а затем удален. Возможна и такая ситуация: достигнув начала очереди, клиент не может получить надлежащего обслуживания и снова возвращается в очередь, после чего обслуживается еще раз.

В программировании схожая ситуация. Приложение добавляет в очередь задание — фрагмент кода, дающий приложению указания, как должно выполняться определенное поведение. После этого другая структура приложения, обычно обработчик очередей, берет на себя задачу последовательного извлечения заданий из очереди и выполнения соответствующего поведения. Обработчик очередей может

удалить задания, возвратить в очередь с некоторой задержкой или пометить как успешно обработанное.

Laravel позволяет легко обслуживать очереди с помощью Redis, *beanstalkd*, Amazon Simple Queue Service (SQS) или таблицы базы данных. Можно применять драйвер `sync`, чтобы задания выполнялись непосредственно в приложении без их фактического помещения в очередь, или драйвер `null`, чтобы задания просто отбрасывались. Оба способа обычно используются в среде локальной разработки и тестирования.

Зачем нужны очереди

Очереди позволяют удалить из любого синхронного вызова ресурсозатратный или медленный процесс. Наиболее распространенным примером является отправка почты — процесс может быть достаточно медленным. Потому лучше не заставлять пользователей ждать, пока завершится отправка почты в ответ на произведенные ими действия. Вместо этого можно поместить в очередь задание «отправка почты», позволив пользователям заняться чем-нибудь другим. Иногда важнее не экономия времени пользователей, а наличие такого процесса, как задание `cron` или веб-хук, включающего в себя достаточно большой объем работы. Вместо того чтобы выполнять этот процесс целиком (и, возможно, исчерпать лимит времени), можно поместить в очередь его отдельные части и позволить обработчику очередей выполнить их по отдельности.

Кроме того, если вы имеете дело с обработкой сложного вида, с которой не может справиться ваш сервер, разверните сразу несколько обработчиков очередей. Это обеспечит более высокую скорость обработки по сравнению с тем, что дает обычный сервер приложений.

Базовая конфигурация очередей

Для очередей предусмотрен отдельный файл конфигурации (`config/queue.php`), в котором можно настроить использование различных драйверов и указать, какой из них должен быть по умолчанию. Там также размещаются аутентификационные данные для SQS, Redis или *beanstalkd*.



Простые очереди на базе Redis в Laravel Forge

Laravel Forge (<http://forge.laravel.com/>) — это предлагаемый Тейлором Отвелом, создателем Laravel, сервис для управления хостингом, который до предела упрощает задачу обслуживания очередей с помощью Redis. Каждый создаваемый вами сервер автоматически настраивается на применение Redis. Чтобы использовать Redis в качестве драйвера очередей, достаточно открыть на любом сайте консоль сервиса Forge, перейти на вкладку Queue (Очередь) и выбрать команду `Start Worker` (Запустить обработчик). Изменять настройки по умолчанию или делать что-то еще не требуется.

Задания в очереди

Еще раз вернемся к аналогии с банком. С точки зрения программирования каждый клиент в банковской *очереди* представляет собой *задание*. В зависимости от среды, эти задания могут принимать самые разные формы: от массива данных до простой строки. В Laravel каждое задание представляет собой коллекцию данных, включающую в себя имя задания, полезную нагрузку, количество уже предпринятых попыток обработки этого задания и другие простые метаданные.

Однако не нужно беспокоиться обо всех этих данных при взаимодействиях с Laravel. Фреймворк предлагает вам структуру с именем *Job* (Задание), которая инкапсулирует отдельную задачу (поведение, команду на выполнение которого возможно выдать приложению) и может добавляться и извлекаться из очереди. Предусмотрены также простые вспомогательные функции, позволяющие легко помещать в очередь команды Artisan и электронные письма.

Для начала рассмотрим пример, в котором каждый раз, когда пользователь изменяет свой тариф на использование вашего SaaS-приложения, нужно заново рассчитывать общую прибыль.

Создание задания

Нужно воспользоваться специальной командой Artisan:

```
php artisan make:job CrunchReports
```

Результат см. в примере 16.1.

Пример 16.1. Стандартный шаблон задания Laravel

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class CrunchReports implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Создание нового экземпляра задания
     *
     * @return void
     */
    public function __construct()
    {
```

```

        //
    }

    /**
     * Выполнение задания
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}

```

5.3 Данный шаблон импортирует трейты `Dispatchable`, `Queueable`, `InteractsWithQueue` и `SerializesModels` и реализует интерфейс `ShouldQueue`. До версии 5.3 часть этой функциональности предоставлялась через родительский класс `App\Jobs`.

Шаблон также содержит два метода: конструктор для передачи данных в задание и метод `handle()`, в котором определяется логика задания (и сигнатура метода для внедрения зависимостей).

Трейты и интерфейс снабжают данный класс возможностью добавления в очередь и взаимодействия с ней. Трейт `Dispatchable` снабжает его методами для отправки заданий. `Queueable` определяет способ помещения задания в очередь. `InteractsWithQueue` при обработке каждого задания дает контроль над его отношениями с очередью, включая удаление и повторное помещение. `SerializesModels` позволяет выполнять сериализацию и десериализацию моделей Eloquent.



Сериализация моделей

Трейт `SerializesModels` снабжает задание способностью сериализовать (преобразовать в «плоский» формат, который можно сохранять в таком хранилище данных, как БД или система очередей) внедряемые модели так, чтобы они были доступны для метода `handle()` задания. Сериализовать весь объект Eloquent слишком сложно, данный трейт сериализует только первичные ключи всех прикрепленных объектов Eloquent при помещении задания в очередь. На этапе десериализации и обработки он извлекает эти модели Eloquent напрямую из базы данных по их первичному ключу. При запуске задания на выполнение оно извлечет «свежий» экземпляр такой модели, а не экземпляр, отражающий ее состояние на момент помещения задания в очередь.

Заполним методы для нашего демонстрационного класса кодом, как показано в примере 16.2.

Пример 16.2. Пример задания

```

...
use App\ReportGenerator;

```

```
class CrunchReports implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
    protected $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function handle(ReportGenerator $generator)
    {
        $generator->generateReportsForUser($this->user);

        Log::info('Generated reports.');
    }
}
```

Мы рассчитываем, что при создании задания будет автоматически внедряться экземпляр `User`. Далее в коде для его обработки размещаем подсказки типа для класса `ReportGenerator` (который мы предположительно создали) и класса `Logger` (предоставляется Laravel). Фреймворк прочитает обе подсказки типа и автоматически внедрит эти зависимости.

Помещение задания в очередь

5.3 Для отправки задания можно использовать много методов, включая доступные для каждого контроллера, и глобальный хелпер `dispatch()`. Но с Laravel 5.5 у нас есть более простой и предпочтительный способ: вызов метода `dispatch()` непосредственно в задании. Таким образом, если вы используете Laravel 5.5 или более новую версию, проигнорируйте другие варианты, как сделаю я в этой главе.

Чтобы отправить задание, необходимо создать экземпляр задания и вызвать в нем метод `dispatch()`, передав ему все необходимые данные (пример 16.3).

Пример 16.3. Отправка заданий

```
$user = auth()->user();
$daysToCrunch = 7;
\App\Jobs\CrunchReports::dispatch($user, $daysToCrunch);
```

Указать, как именно должно отправляться задание, можно с помощью таких трех параметров, как подключение, очередь и задержка.

Настройка подключения. При наличии сразу нескольких подключений к очереди можно указать конкретное подключение, пристыковав к методу `dispatch()` метод `onConnection()`:

```
DoThingJob::dispatch()->onConnection('redis');
```

Настройка очереди. В случае использования сервера очередей можно указать, в какую именованную очередь следует поместить задание. Например, вы можете различать задания по уровню важности, используя две очереди с названиями `low` (низкий) и `high` (высокий).

В какую очередь должно помещаться задание, указывается методом `onQueue()`:

```
DoThingJob::dispatch()->onQueue('high');
```

Настройка задержки. С помощью метода `delay()` можно задать длительность задержки, выдерживаемой обработчиками очередей перед обработкой задания. Данный метод принимает либо целое число, представляющее длительность задержки в секундах, либо экземпляр класса `DateTime/Carbon`:

```
// Задержка на пять минут перед «освобождением» задания для обработчиков очередей
$delay = now()->addMinutes(5);
DoThingJob::dispatch()->delay($delay);
```

Обратите внимание, что сервис Amazon SQS не позволяет использовать задержки, превышающие 15 минут.

Запуск обработчика очередей

Что представляет собой обработчик очередей и как он работает? В Laravel эту роль играет команда `Artisan`, которая действует до бесконечности (ее можно остановить вручную), обеспечивая извлечение заданий из очереди и их выполнение:

```
php artisan queue:work
```

Эта команда запускает демон, который «прослушивает» вашу очередь. При наличии в ней заданий он извлекает первое, обрабатывает и удаляет из очереди, после чего переходит к следующему. Если заданий нет, он переходит в состояние «сна» и находится в нем в течение настраиваемого периода времени, потом снова проверяет наличие заданий.

Вы можете указать, сколько секунд должно выполняться задание до его остановки слушателем очереди (`--timeout`), как долго слушатель должен находиться в состоянии «сна» при отсутствии заданий (`--sleep`), количество попыток выполнения задания до его удаления из очереди (`--tries`), какое подключение нужно прослушивать обработчику (первый параметр после `queue:work`) и какие очереди прослушивать (`--queue=`):

```
php artisan queue:work redis --timeout=60 --sleep=15 --tries=3
--queue=high,medium
```

Можно обработать только одно задание с помощью команды `php artisan queue:work`.

Обработка ошибок

Теперь поговорим, что происходит, когда в ходе обработки задания что-то идет не так.

Исключения на этапе обработки

Если выбрасывается исключение, слушатель очереди «выпускает» задание назад в очередь. Так будет раз за разом до успешного выполнения или исчерпания лимита на количество попыток, установленного для вашего слушателя очереди.

Ограничение количества попыток

Максимальное количество попыток задается параметром `--tries`, передаваемым команде `Artisan queue:listen` или `queue:work`.



Опасность бесконечного выполнения повторных попыток

Если параметр `--tries` не задан или задан равным 0, слушатель очереди позволяет предпринимать бесконечное количество попыток. Если задание просто не может быть успешно выполнено (например, ему нужно использовать удаленное сообщение из сети Twitter), ваше приложение будет пытаться до бесконечности, пока не прекратит свою работу.

И документация фреймворка (<http://bit.ly/2TWQHpq>), и сервис Laravel Forge рекомендуют 3 в качестве отправной точки для максимального количества повторных попыток. Если вы не можете выбрать значение, начните с этого и при необходимости увеличьте/уменьшите его:

```
php artisan queue:work --tries=3
```

Если нужно проверить количество предпринятых попыток выполнения задания, вызовите метод `attempts()` в объекте задания (пример 16.4).

Пример 16.4. Проверка в отношении того, сколько было предпринято попыток выполнения задания

```
public function handle()
{
    ...
    if ($this->attempts() > 3) {
        //
    }
}
```

Обработка неудачно выполненных заданий

После превышения допустимого количества повторных попыток задание считается неудачно выполненным. Прежде чем делать что-либо еще — даже если цель лишь ограничение количества повторных попыток, — нужно создать таблицу базы данных для неудачно выполненных заданий.

Затем запустить миграцию с помощью соответствующей команды Artisan:

```
php artisan queue:failed-table
php artisan migrate
```

В эту таблицу будут заноситься все задания, превысившие максимально допустимое количество повторных попыток. Однако с неудачно выполненными заданиями можно делать много чего еще.

К примеру, определить в классе задания метод `failed()`, выполняемый при неудачном выполнении задания (пример 16.5).

Пример 16.5. Определение метода, выполняемого при неудачном выполнении задания

```
...
class CrunchReports implements ShouldQueue
{
    ...

    public function failed()
    {
        // Выполняем любые необходимые действия, например уведомляем администратора
    }
}
```

Или зарегистрировать глобальный обработчик для неудачно выполненных заданий. Для этого определите слушатель, разместив код из примера 16.6 где-нибудь в коде начальной загрузки приложения, — если вы затрудняетесь с выбором подходящего места, расположите его в метод `boot()` провайдера `AppServiceProvider`.

Пример 16.6. Регистрация глобального обработчика для неудачно выполненных заданий

```
// Некоторый сервис-провайдер
use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobFailed;
...
public function boot()
{
    Queue::failing(function (JobFailed $event) {
        // $event->connectionName
        // $event->job
        // $event->exception
    });
}
```

В вашем распоряжении также ряд команд Artisan для взаимодействия с таблицей неудачно выполненных заданий. Команда `queue:failed` отображает список этих заданий.

```
php artisan queue:failed
```

Он будет выглядеть примерно так:

```
+-----+-----+-----+-----+-----+
| ID | Connection | Queue | Class | Failed At |
+-----+-----+-----+-----+-----+
| 9 | database | default | App\Jobs\AlwaysFails | 2018-08-26 03:42:55 |
+-----+-----+-----+-----+-----+
```

Взяв из этого списка идентификатор любого отдельного задания, попробуйте выполнить его повторно с помощью команды `queue:retry`:

```
php artisan queue:retry 9
```

Чтобы попытаться выполнить повторно такие задания, передайте вместо идентификатора строку `all`:

```
php artisan queue:retry all
```

Для удаления отдельного неудачно выполненного задания воспользуйтесь командой `queue:forget`:

```
php artisan queue:forget 5
```

Удалить все можно с помощью команды `queue:flush`:

```
php artisan queue:flush
```

Управление очередью

Иногда в коде для обработки задания требуется разместить условия, на основании которых задание будет «выпускаться» назад в очередь для его повторного выполнения в дальнейшем или безвозвратно удаляться.

«Выпустить» задание назад в очередь можно с помощью метода `release()`, как показано в примере 16.7.

Пример 16.7. «Выпускание» задания назад в очередь

```
public function handle()
{
    ...
    if (condition) {
        $this->release($numberOfSecondsToDelayBeforeRetrying);
    }
}
```

Если нужно удалить задание на этапе его обработки, вызовите в любом месте оператор `return`, как показано в примере 16.8. Очередь воспримет это как сигнал: задание было надлежащим образом обработано, его не нужно возвращать в очередь.

Пример 16.8. Удаление задания

```
public function handle()
{
    ...
    if ($jobShouldBeDeleted) {
        return;
    }
}
```

Очереди для поддержки других функций

Хотя очереди помогают поочередно выполнять задания, еще можно помещать в очередь электронные письма, используя метод `Mail::queue()`, или команды Artisan, о чем мы подробно говорили в главе 8. Подробнее об этом написано в подразделе «Очереди» на с. 426.

Laravel Horizon

Laravel Horizon, как и Scout, Passport и т. д., относится к числу пакетов Laravel, не поставляемых вместе с его ядром.

Пакет Horizon предоставляет информацию о состоянии вашей очереди заданий на базе Redis. Вы можете просматривать сведения о том, какие задания выполнены неудачно, сколько содержится в очереди и насколько быстро производится их обработка. Можно даже получать уведомления о перегрузке и сбоях очередей. Как выглядит информационная панель пакета Horizon, можно увидеть на рис. 16.1.

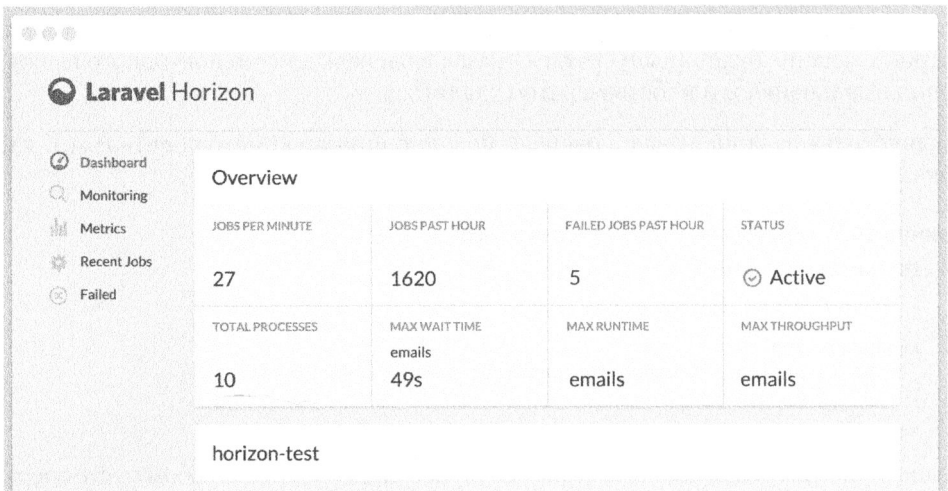


Рис. 16.1. Информационная панель пакета Horizon

Как установить и запустить пакет Horizon, подробно описано в документации. Если интересно, ознакомьтесь с тем, как производится установка, настройка и развертывание пакета Horizon, в его документации по адресу <https://laravel.com/docs/horizon>.

 Для запуска пакета Horizon нужны Laravel 5.5 и PHP 7.1 или более новые версии.

События

В случае заданий вызывающий код информирует приложение о необходимости *что-то сделать*: `CrunchReports` (Составить отчеты) или `NotifyAdminOfNewSignup` (Уведомить администратора о новой регистрации).

Касательно событий он сообщает приложению, что *что-то произошло*: `UserSubscribed` (Пользователь подписался), `UserSignedUp` (Пользователь зарегистрировался), `ContactWasAdded` (Добавлен контакт). События — это уведомление о том, что что-то случилось.

Одни из этих событий могут быть запущены самим фреймворком. Например, модели Eloquent запускают события в момент их сохранения, создания и удаления. Вторые возможно запустить вручную из кода приложения.

Само запущенное событие ничего не делает. Однако можно привязать к нему *слушатели событий*, единственное назначение которых — прослушивание трансляции конкретных событий и запуск ответных действий. Любое событие может иметь любое количество слушателей либо вообще не иметь таковых.

Структура событий в Laravel соответствует шаблону публикации/подписки, или, иначе говоря, шаблону наблюдателя. Многие события транслируются в приложение, при этом какие-то из них вообще не прослушиваются, а некоторые имеют десятки слушателей. Для самих событий это неважно.

Запуск события

Запустить событие можно тремя способами: использовать фасад `Event` или глобальный хелпер `event()`, внедрить класс `Dispatcher`, как показано в примере 16.9.

Пример 16.9. Три способа запуска события

```
Event::fire(new UserSubscribed($user, $plan));
// или
$dispatcher = app(Illuminate\Contracts\Events\Dispatcher::class);
$dispatcher->fire(new UserSubscribed($user, $plan));
// или
event(new UserSubscribed($user, $plan));
```

Я рекомендую воспользоваться глобальным хелпером.

Чтобы создать запускаемое событие, воспользуйтесь командой `Artisan make:event`:

```
php artisan make:event UserSubscribed
```

Эта команда сгенерирует файл, который будет выглядеть примерно так, как в примере 16.10.

Пример 16.10. Шаблон события, создаваемый Laravel по умолчанию

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class UserSubscribed
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Создание нового экземпляра события
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Получение каналов, по которым должно транслироваться событие
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

Что мы здесь имеем? Трейт `SerializesModels` работает так же, как и в случае заданий, давая возможность передачи моделей Eloquent в качестве параметров. Трейт `InteractsWithSockets`, интерфейс `ShouldBroadcast` и метод `broadcastOn()` обеспечивают функциональность трансляции событий с помощью веб-сокетов (WebSockets), о которой мы поговорим чуть позже.

Здесь нет метода `handle()` (обработать) или `fire()` (запустить), потому что данный объект не предназначен для определения конкретного действия. Он должен лишь инкапсулировать некоторые данные. Первый фрагмент данных — имя события. Так, название `UserSubscribed` говорит, что произошло конкретное событие (пользователь подписался). Остальная часть — любые данные, передаваемые в конструктор и связываемые с конкретной сущностью.

Пример 16.11 показывает, как мы могли бы отредактировать этот код в случае события `UserSubscribed`.

Пример 16.11. Внедрение данных в событие

```
...
class UserSubscribed
{
    use InteractsWithSockets, SerializesModels;

    public $user;
    public $plan;

    public function __construct($user, $plan)
    {
        $this->user = $user;
        $this->plan = $plan;
    }
}
```

Теперь у нас есть объект, который надлежащим образом представляет произошедшее событие: пользователь `$event->user` подписался на тарифный план `$event->plan`. Чтобы запустить это событие, достаточно вызвать метод `event(new UserSubscribed($user, $plan))`.

Прослушивание события

Таким образом, у нас уже есть событие и мы знаем, как его запустить. Теперь посмотрим, как обеспечить его прослушивание.

Необходимо создать слушатель событий. Предположим, что нам нужно отправлять электронное письмо владельцу приложения каждый раз, когда новый пользователь оформляет подписку:

```
php artisan make:listener EmailOwnerAboutSubscription --event=UserSubscribed
```

Эта команда сгенерирует файл, показанный в примере 16.12.

Пример 16.12. Шаблон слушателя событий, создаваемый фреймворком Laravel по умолчанию

```
<?php

namespace App\Listeners;
```

```

use App\Events\UserSubscribed;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailOwnerAboutSubscription
{
    /**
     * Создание слушателя событий
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Обработка события
     *
     * @param UserSubscribed $event
     * @return void
     */
    public function handle(UserSubscribed $event)
    {
        //
    }
}

```

Вот, оказывается, где выполняются действия и находится метод `handle()`. Этот метод принимает событие типа `UserSubscribed` и работает соответственно.

Заставим его отправлять электронное письмо (пример 16.13).

Пример 16.13. Пример слушателя событий

```

...
use App\Mail\UserSubscribed as UserSubscribedMessage;

class EmailOwnerAboutSubscription
{
    public function handle(UserSubscribed $event)
    {
        Log::info('Emailed owner about new user: ' . $event->user->email);

        Mail::to(config('app.owner-email'))
            ->send(new UserSubscribedMessage($event->user, $event->plan));
    }
}

```

Теперь нужно настроить его на прослушивание события `UserSubscribed`. Это можно сделать в свойстве `$listen` класса `EventServiceProvider` (пример 16.14).

Пример 16.14. Привязка слушателей к событиям в провайдере EventServiceProvider

```
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        \App\Events\UserSubscribed::class => [
            \App\Listeners\EmailOwnerAboutSubscription::class,
        ],
    ];
}
```

Ключ каждого элемента массива представляет собой имя класса события, а значение — массив названий классов слушателей. Можно указать любое количество имен классов для ключа `UserSubscribed`, и все они будут прослушивать и реагировать на каждое событие `UserSubscribed`.

Подписчики событий

Laravel предлагает вам еще одну структуру для определения связей между событиями и их слушателями. Это так называемый *подписчик событий* — класс, который содержит набор методов в качестве отдельных слушателей уникальных событий вместе с определением того, какие события должен обрабатывать тот или иной метод. Взгляните на код примера 16.15. Примите к сведению, что подписчики событий не очень распространенный инструмент.

Пример 16.15. Пример подписчика событий

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    public function onUserSubscription($event)
    {
        // Обрабатывает событие UserSubscribed
    }

    public function onUserCancellation($event)
    {
        // Обрабатывает событие UserCanceled
    }

    public function subscribe($events)
    {
        $events->listen(
            \App\Events\UserSubscribed::class,
            'App\Listeners\UserEventSubscriber@onUserSubscription'
        );

        $events->listen(
```

```

        \App\Events\UserCanceled::class,
        'App\Listeners\UserEventSubscriber@onUserCancellation'
    );
}
}

```

В подписчике должен быть определен метод `subscribe()`, принимающий экземпляр диспетчера событий. Данный метод служит для сопоставления событий с их слушателями, в качестве которых здесь выступают методы данного класса, а не целые классы.

Еще раз напомним: если вы видите в коде символ `@`, то слева от него находится имя класса, а справа — имя метода. То есть в примере 16.15 мы указываем, что метод `onUserSubscription()` данного подписчика должен прослушивать события `UserSubscribed`.

После этого остается в провайдере `App\Providers\EventServiceProvider` добавить имя класса нашего подписчика в свойство `$subscribe`, как показано в примере 16.16.

Пример 16.16. Регистрация подписчика событий

```

...
class EventServiceProvider extends ServiceProvider
{
    ...
    protected $subscribe = [
        \App\Listeners\UserEventSubscriber::class
    ];
}

```

Трансляция событий посредством веб-сокетов и Laravel Echo

Веб-сокеты (WebSocket, WebSockets) — это протокол, пропагандируемый компанией Pusher (создавшей одноименный SaaS-сервис на базе веб-сокетов). Он обеспечивает коммуникацию между веб-устройствами практически в режиме реального времени. Вместо того чтобы передавать данные посредством HTTP-запросов, библиотеки WebSockets устанавливают прямое соединение между клиентом и сервером. На базе веб-сокетов построены инструменты вроде окон чатов в Gmail и Facebook, где не нужно ждать перезагрузки страницы или пока Ajax-запросы получают/отправляют данные. Вместо этого данные отправляются и принимаются в режиме реального времени.

Веб-сокеты лучше всего подходят для небольших фрагментов данных, передаваемых в формате публикации/подписки, таких как события Laravel. В состав фреймворка включен набор инструментов, позволяющих легко указать, что одно

или несколько ваших событий должны транслироваться на сервер веб-сокетов. Например, можно обеспечить публикацию события `MessageWasReceived` (Сообщение получено) в окне уведомлений определенного пользователя или группы пользователей при поступлении сообщения в приложение.

LARAVEL ECHO

В составе Laravel имеется инструмент мощнее, предназначенный для более сложной трансляции событий. Если нужно обеспечить уведомление о присутствии в сети или синхронизировать насыщенную модель данных клиентского интерфейса со своим приложением Laravel, обратите внимание на пакет Laravel Echo. Его рассмотрим в подразделе «Продвинутые инструменты трансляции» на с. 460. Хотя многие из возможностей пакета Echo встроены в ядро фреймворка, некоторые его возможности все же требуют загрузки внешней JavaScript-библиотеки Echo, о которой мы поговорим в подразделе «Laravel Echo (сторона JavaScript-кода)» на с. 464.

Конфигурация и настройка

Параметры конфигурации для трансляции событий находятся в файле `config/broadcasting.php`. Laravel позволяет использовать для трансляции следующие три драйвера: платного SaaS-сервиса Pusher, хранилища Redis для локально работающих серверов веб-сокетов и `log` для локальной разработки и отладки.



Слушатели очередей

Для быстрой трансляции событий Laravel помещает соответствующие инструкции трансляции в очередь. Вам потребуется запустить обработчик очередей (или использовать драйвер очередей `sync` для локальной разработки). Как запускается обработчик очередей, рассказано в подразделе «Запуск обработчика очередей» на с. 444.

По умолчанию Laravel настраивает обработчики очередей на повторную проверку наличия новых заданий с задержкой 3 секунды. Однако в случае трансляции событий определенные события транслируются с интервалом 1–2 секунды. Можно ускорить этот процесс, изменив настройки очереди так, чтобы величина задержки перед повторной проверкой на наличие новых заданий составляла только секунду.

Трансляция события

Чтобы обеспечить трансляцию события, необходимо пометить его как транслируемое событие, реализовав в нем интерфейс `Illuminate\Contracts\Broadcasting\ShouldBroadcast`. Он требует наличия метода `broadcastOn()`, возвращающего массив строк или объектов `Channel`, представляющих собой WebSocket-каналы.

СТРУКТУРА WEBSOCKET-СОБЫТИЙ

Каждое событие, отправляемое посредством веб-сокетов, может иметь три основные характеристики: имя, канал и данные.

В качестве *имени* события подходит строка вида `user-was-subscribed`, но по умолчанию Laravel использует полное название класса события, скажем, `App\Events\UserSubscribed`. Можно задать пользовательское имя, передав его в качестве параметра необязательному методу `broadcastAs()` в классе события.

Канал представляет собой способ определения того, какие клиенты должны получать данное сообщение. Очень распространенный шаблон — создать отдельные каналы для каждого пользователя (например, `users.1`, `users.2` и т. д.) и, возможно, еще канал для всех пользователей (например, `users`) и членов определенной учетной записи (`accounts.1`).

Имя частного канала должно начинаться с префикса `private-`, а канала присутствия — с префикса `presence-`. То есть частный канал сервиса Pusher следует назвать не `groups.5`, а `private-groups.5`. Вы можете обеспечить автоматическое добавление этих префиксов к именам каналов, используя предоставляемые Laravel объекты `PrivateChannel` и `PresenceChannel` в методе `broadcastOn()`.

Что такое каналы присутствия и публичные/приватные каналы, см. в примечаниях в пункте «Сервис-провайдер трансляции» на с. 462.

В качестве *данных* здесь выступает полезная нагрузка, обычно в формате JSON, из относящейся к событию информации, например сообщение, данные о пользователе или тарифном плане, над которыми можно произвести действия путем запуска JavaScript-кода.

В примере 16.17 показано, как модифицировать код события `UserSubscribed` для его трансляции по двум каналам: по одному для пользователя (подтверждение его подписки) и одному для администраторов (уведомление их о новой подписке).

Пример 16.17. Событие, транслируемое по нескольким каналам

```
...
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class UserSubscribed implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $user;
    public $plan;

    public function __construct($user, $plan)
    {
        $this->user = $user;
```



```
        $this->plan = $plan;
    }

    public function broadcastOn()
    {
        // Использование строк
        return [
            'users.' . $this->user->id,
            'admins'
        ];

        // Использование объектов Channel
        return [
            new Channel('users.' . $this->user->id),
            new Channel('admins'),
            // В случае приватного канала: new PrivateChannel('admins'),
            // В случае канала присутствия: new PresenceChannel('admins'),
        ];
    }
}
```

По умолчанию все публичные свойства события сериализуются в формат JSON и отправляются вместе с транслируемым событием как его данные. То есть данные транслируемых нами событий `UserSubscribed` могут выглядеть так, как показано в примере 16.18.

Пример 16.18. Пример данных транслируемого события

```
{
    'user': {
        'id': 5,
        'name': 'Fred McFeely',
        ...
    },
    'plan': 'silver'
}
```

Переопределить это поведение поможет возвращение массива данных из метода `broadcastWith()` события (пример 16.19).

Пример 16.19. Настройка данных транслируемого события

```
public function broadcastWith()
{
    return [
        'userId' => $this->user->id,
        'plan' => $this->plan
    ];
}
```

Можно указать, в какую очередь должно помещаться событие, задав свойство `$broadcastQueue` в классе события:

```
public $broadcastQueue = 'websockets-for-faster-processing';
```

Обычно это нужно, чтобы трансляцию событий не замедляли другие элементы очереди. В использовании веб-сокетов реального времени мало смысла, если события будут отправляться с задержкой из-за наличия перед ними медленно выполняющихся заданий.

Возможно не помещать событие в очередь (чтобы использовался драйвер очереди sync, который обрабатывается в текущем потоке PHP), реализовав в нем контракт `ShouldBroadcastNow` (пример 16.20).

Пример 16.20. Отправка события в обход очереди трансляции

```
use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;
```

```
class UserSubscribed implements ShouldBroadcastNow
{
    //
}
```

Можно указать, в каких случаях следует транслировать событие, а в каких — нет, снабдив его методом `broadcastWhen()` (пример 16.21).

Пример 16.21. Условная трансляция события

```
public function broadcastWhen()
{
    // Уведомление только в случае оформления подписки пользователями из Белого дома
    return str_contains($this->user->email, 'whitehouse.gov');
}
```

Получение сообщения

Если вы решили использовать собственный сервер веб-сокетов на базе Redis, то в документации фреймворка Laravel (<http://bit.ly/2VApJBb>) есть прекрасное пошаговое руководство по настройке такого сервера с использованием библиотек `socket.io` и `ioredis`.

На момент публикации этой книги Laravel-разработчики чаще всего применяли для этой цели сервис Pusher (<https://pusher.com/>). Хотя при большом количестве подключений сервис платный, но имеется и бесплатный тарифный план. Pusher дает легко настроить простой сервер веб-сокетов, а его комплект средств разработки на языке JavaScript (SDK) позволяет реализовать любое управление аутентификацией и каналами практически без усилий. Предлагаются версии SDK для IOS, Android и многих других платформ, языков и фреймворков.

Незадолго до публикации данной книги было объявлено о выходе нового инструмента с названием Laravel WebSockets (<http://bit.ly/2HS4rur>), который предназначен для создания собственного сервера веб-сокетов на базе Laravel с совместимостью

с сервисом Pusher. Этот пакет можно установить в текущее приложение Laravel (из которого вы осуществляете трансляцию) или в отдельный микросервис.

Если вы решили настроить сервер с помощью Laravel WebSockets, то следуйте приведенным здесь указаниям для использования сервиса Pusher. В вашем случае будут лишь немного различаться параметры конфигурации.

Простейший способ прослушивания веб-сокетов с помощью сервиса Pusher

Даже если в итоге вы решите использовать Echo, будет нелишним узнать, как в Laravel прослушивать транслируемые события без этого пакета. Значительная часть этого кода не нужна при использовании Echo, я рекомендую после прочтения этого подраздела ознакомиться также с подразделом «Laravel Echo (сторона JavaScript-кода)» на с. 464. Так вы сможете выбрать подходящий вариант.

Для начала подгрузите библиотеку сервиса Pusher, получите ключ API-интерфейса из вашей учетной записи сервиса Pusher и подпишитесь на все события во всех каналах, как показано в примере 16.22.

Пример 16.22. Простейший способ использования сервиса Pusher

```
...
<script src="https://js.pusher.com/4.3/pusher.min.js"></script>
<script>
// Включаем логирование сервиса Pusher – исключите эту строку
// для среды эксплуатации
Pusher.logToConsole = true;

// Вероятно, в глобальной области видимости; это лишь пример того,
// как можно получить данные
var App = {
  'userId': {{ auth()->id() }},
  'pusherKey': '{{ config('broadcasting.connections.pusher.key') }}'
};

// В локальной области видимости
var pusher = new Pusher(App.pusherKey, {
  cluster: '{{ config('broadcasting.connections.pusher.options.cluster') }}',
  encrypted: {{ config('broadcasting.connections.pusher.options.encrypted') }}
});

var pusherChannel = pusher.subscribe('users.' + App.userId);

pusherChannel.bind('App\\Events\\UserSubscribed', (data) => {
  console.log(data.user, data.plan);
});
</script>
```



Модификация символов обратной косой черты в JavaScript

\ — управляющий символ в языке JavaScript, и потому должен дублироваться для его представления в строках: \\. Именно этим объясняется то, что в примере 16.22 мы удвоили каждый символ обратной косой черты, разделяющий сегменты пространства имен.

Для публикации в сервис Pusher из Laravel возьмите с информационной панели учетной записи сервиса Pusher свой ключ сервиса Pusher, пароль, кластер и идентификатор приложения. Укажите эти данные в файле `.env`, используя ключи `PUSHER_KEY`, `PUSHER_SECRET`, `PUSHER_APP_CLUSTER` и `PUSHER_APP_ID`.

Если вы раздаете свое приложение, то перейдите на страницу с JavaScript-кодом из примера 16.22 в одном окне браузера и запустите транслируемое событие в другом окне браузера или окне терминала. Проследите, чтобы при этом работал слушатель очередей или использовался драйвер `sync` и были должным образом настроены аутентификационные данные. После этого в окне консоли JavaScript начнут выводиться в режиме реального времени протокольные сообщения о событиях.

Имея в своем распоряжении такую мощную возможность, можно легко информировать пользователей о любых изменениях в их данных при каждом входе в приложение. Или уведомлять их о действиях других пользователей, завершении длительных процессов или действиях приложения в ответ на такие внешние воздействия, как поступление электронных писем или срабатывание веб-хуков — практически о чем угодно.



Требования

Для обеспечения трансляции с использованием Pusher или Redis потребуется подгрузить следующие зависимости.

- Pusher: `pusher/pusher-php-server ~3.0`.
- Redis: `redis/predis`.

Продвинутые инструменты трансляции

Laravel предлагает еще несколько инструментов для более сложных взаимодействий при трансляции событий. Они выделены в пакет *Laravel Echo*, который представляет собой комбинацию из компонентов фреймворка и JavaScript-библиотеки.

Компоненты *Laravel Echo* лучше всего использовать в JavaScript-коде клиентского интерфейса (о чем будет рассказано в подразделе «Laravel Echo (сторона JavaScript-кода)» на с. 464). Однако некоторые из возможностей этого пакета мож-

но реализовать и без использования JavaScript-компонентов. Хотя Echo позволяет использовать и сервис Pusher, и хранилище Redis, я приведу здесь примеры только для первого случая.

Исключение текущего пользователя из числа получателей транслируемых событий

Каждому подключению к сервису Pusher присваивается уникальный идентификатор сокета. Можно указать, что определенный сокет (пользователь) должен быть исключен из числа получателей конкретного транслируемого события.

Это позволяет обозначить, что определенные события не должны транслироваться запустившему их пользователю. Допустим, каждый член какой-то команды уведомляется о создании новых задач другими пользователями. Нужно ли в таком случае сообщать пользователю, что он только что сам создал новую задачу? Конечно, нет, и поэтому в Echo есть метод `toOthers()`.

Такое исключение реализуется в два этапа. Сначала следует настроить JavaScript-код так, чтобы он отправлял определенный POST-запрос сокету `/broadcasting/socket` при инициализации WebSocket-подключения. Это обеспечит присвоение идентификатора сокета `socket_id` вашей Laravel-сессии. Пакет Echo берет эту задачу на себя, но вы можете сделать это и вручную — взгляните на исходный код пакета Echo (<http://bit.ly/2CAM89w>).

Затем нужно снабдить каждый создаваемый JavaScript-кодом запрос заголовком `X-Socket-ID`, содержащим идентификатор сокета `socket_id`. Пример 16.23 показывает, как это можно сделать с использованием Axios или JQuery. Чтобы вызвать метод `toOthers()`, событие должно применять трейт `Illuminate\Broadcasting\InteractsWithSockets`.

Пример 16.23. Отправка идентификатора сокета в каждом Ajax-запросе с использованием Axios или JQuery

```
// Запускается сразу после инициализации пакета Echo
// с использованием Axios
window.axios.defaults.headers.common['X-Socket-Id'] = Echo.socketId();

// С использованием jquery
$.ajaxSetup({
  headers: {
    'X-Socket-Id': Echo.socketId()
  }
});
```

После этого можно исключить запустившего событие пользователя из числа его получателей путем вызова глобального хелпера `broadcast()` вместо глобального хелпера `event()`, с пристыковкой к нему метода `toOthers()`:

```
broadcast(new UserSubscribed($user, $plan))->toOthers();
```

Сервис-провайдер трансляции

Остальные возможности пакета Echo требуют, чтобы JavaScript-код выполнил аутентификацию для доступа к серверу. Взгляните на код провайдера `App\Providers\BroadcastServiceProvider`, где обычно определяется способ авторизации пользователей для доступа к приватным каналам и каналам присутствия.

Здесь есть две основные возможности: определить параметры авторизации для каналов и `middleware`, которое будет вызываться при выборе аутентификационных маршрутов трансляции.

Чтобы использовать эти возможности, потребуется раскомментировать строку `App\Providers\BroadcastServiceProvider::class` в файле `config/app.php`.

Если вы решите использовать эти возможности *без* Laravel Echo, нужно либо вручную реализовать отправку CSRF-токенов в аутентификационных запросах, либо исключить маршруты `/broadcasting/auth` и `/broadcasting/socket` из числа маршрутов, подлежащих CSRF-защите, добавив их в свойство `$except middleware VerifyCsrfToken`.

Привязка определений авторизации к каналам веб-сокетов. Приватные каналы и каналы присутствия веб-сокетов должны иметь возможность отправлять приложению запросы в отношении того, авторизован ли пользователь для доступа к этому каналу. Правила такой авторизации определяются с помощью метода `Broadcast::channel()` в файле `routes/channels.php`.



Публичные каналы, приватные каналы и каналы присутствия

При использовании веб-сокетов доступны три типа каналов: публичные, приватные и каналы присутствия.

На *публичные каналы* может подписаться любой пользователь вне зависимости от того, прошел он аутентификацию или нет.

Приватные требуют, чтобы JavaScript-код конечного пользователя выполнил аутентификацию в отношении приложения, подтверждающую, что пользователь и аутентифицирован, и авторизован для присоединения к этому каналу.

Каналы присутствия — это разновидность приватных каналов, где вместо передачи сообщений отслеживается, какие пользователи присоединяются к каналу/отсоединяются от него, и эта информация становится доступной для клиентского интерфейса приложения.

Метод `Broadcast::channel()` принимает два параметра. Первым параметром передается строка, указывающая, к какому каналу должен применяться метод. Вторым — замыкание, определяющее способ авторизации пользователей для доступа к каналам, совпавшим с переданной строкой. Замыкание принимает в качестве первого параметра модель Eloquent текущего пользователя, а дополнительных

параметров — совпавшие сегменты `variableNameHere`. Например, при применении определения авторизации канала со строкой `teams.teamId` к каналу `teams.5` его замыканию будет передан объект `$user` как первый параметр и `5` — как второй.

При определении правил для приватного канала замыкание метода `Broadcast::channel()` должно возвращать логическое значение, указывающее, авторизован пользователь для этого канала или нет. При определении правил для канала присутствия это замыкание должно возвращать массив, указывающий, какие данные о присутствующем в канале пользователе нужно сделать доступными для канала присутствия (пример 16.24).

Пример 16.24. Определение правил авторизации для приватных каналов и каналов присутствия веб-сокетов

```
...
// routes/channels.php

// Определяем способ аутентификации приватного канала
Broadcast::channel('teams.{teamId}', function ($user, $teamId) {
    return (int) $user->team_id === (int) $teamId;
});

// Определяем способ аутентификации приватного канала; возвращаем те данные
// о пользователе в канале, которые должно иметь приложение
Broadcast::channel('rooms.{roomId}', function ($user, $roomId) {
    if ($user->rooms->contains($roomId)) {
        return [
            'name' => $user->name
        ];
    }
});
```

Каким образом эта информация попадает из приложения Laravel в JavaScript-код клиентского интерфейса? JavaScript-библиотека сервиса Pusher отправляет приложению POST-запрос? По умолчанию он направляется по маршруту `/pusher/auth`, но можно перенаправить его на маршрут аутентификации Laravel — `/broadcasting/auth` (и пакет Echo возьмет эту задачу на себя):

```
var pusher = new Pusher(App.pusherKey, {
    authEndpoint: '/broadcasting/auth'
});
```

В примере 16.25 показано, как видоизменить код для приватных каналов и каналов присутствия из примера 16.22, если вы решите обойтись без клиентских компонентов пакета Echo.

Пример 16.25. Простейший способ использования сервиса Pusher для приватных каналов и каналов присутствия

```
...
<script src="https://js.pusher.com/4.3/pusher.min.js"></script>
<script>
```

```

// Включаем логирование сервиса Pusher — исключите эту строку
// для среды эксплуатации
Pusher.logToConsole = true;

// Вероятно, в глобальной области видимости; это лишь пример,
// как можно получить данные
var App = {
  'userId': {{ auth()->id() }},
  'pusherKey': '{{ config('broadcasting.connections.pusher.key') }}'
};

// В локальной области видимости
var pusher = new Pusher(App.pusherKey, {
  cluster: '{{ config('broadcasting.connections.pusher.options.cluster') }}',
  encrypted: {{ config('broadcasting.connections.pusher.options.encrypted') }}
}),
  authEndpoint: '/broadcasting/auth'
});

// Приватный канал
var privateChannel = pusher.subscribe('private-teams.1');
privateChannel.bind('App\\Events\\UserSubscribed', (data) => {
  console.log(data.user, data.plan);
});

// Канал присутствия
var presenceChannel = pusher.subscribe('presence-rooms.5');

console.log(presenceChannel.members);
</script>

```

Что ж, теперь мы можем отправлять WebSocket-сообщения пользователям в зависимости от того, выполнили ли они правила авторизации заданного канала. Мы также можем отслеживать, кто активен в определенной группе пользователей или в определенной части сайта, и отображать каждому пользователю соответствующую информацию о других членах той же группы.

Laravel Echo (сторона JavaScript-кода)

Laravel Echo состоит из двух частей — продвинутых возможностей фреймворка, которые мы только что рассмотрели, и JavaScript-пакета, использующего все эти возможности и позволяющего вам писать гораздо меньше стереотипного кода для создания мощных клиентских интерфейсов на базе веб-сокетов. JavaScript-пакет Echo позволяет легко осуществлять аутентификацию, авторизацию и подписку на приватные каналы и каналы присутствия. Echo можно использовать в сочетании с комплектами средств разработки (SDK) для сервиса Pusher (в случае применения сервиса Pusher или совместимого с ним пользовательского сервера) и библиотеки `socket.io` (в случае использования хранилища Redis).

Загрузка пакета Echo в проект

Для использования пакета Echo в JavaScript-коде своего проекта добавьте его в файл `package.json` с помощью команды `npm install --save`. Не забудьте также подгрузить соответствующий SDK для сервиса Pusher или библиотеки `socket.io`:

```
npm install pusher-js laravel-echo --save
```

Предположим, у вас есть простейший файл Laravel Mix, который компилирует ваш файл `app.js` с помощью Webpack, как показано в примере 16.26.

Пример 16.26. Компиляция файла `app.js` с помощью Laravel Mix

```
let mix = require('laravel-mix');

mix.js('resources/assets/js/app.js', 'public/js');
```

Структура файла `resources/js/app.js`, предлагаемая Laravel по умолчанию, — отличный пример надлежащей инициализации пакета Echo после его установки. В примере 16.27 показано, как осуществляется эта инициализация в файлах `resources/js/app.js` и `resources/js/bootstrap.js`.

Пример 16.27. Инициализация пакета Echo в файлах `app.js` и `bootstrap.js`

```
// app.js
require('./bootstrap');

// ... множество Vue-компонентов ...

// Добавьте здесь привязки пакета Echo

// bootstrap.js
import Echo from "laravel-echo";

window.Echo = new Echo({
  broadcaster: 'pusher',
  key: process.env.MIX_PUSHER_APP_KEY,
  cluster: process.env.MIX_PUSHER_APP_CLUSTER
});
```

Для обеспечения CSRF-защиты нужно добавить в HTML-шаблоне тег `csrf-token` `<meta>`:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Не забудьте добавить в HTML-шаблоне ссылку на откомпилированный файл `app.js`:

```
<script src="{{ asset('js/app.js') }}"></script>
```

После этого уже можно будет приступить к работе.



Различия в конфигурации при использовании серверного пакета **Laravel WebSockets**

Если вы используете сервер **Laravel WebSockets** (о его применении рассказано в подразделе «Получение сообщения» на с. 458), то ваши настройки конфигурации будут немного отличаться от того, что показано в примере 16.27. Более подробные сведения см. в документации по пакету **Laravel WebSockets** (<http://bit.ly/2Txh2Wv>).

Использование **Echo** для простейшей трансляции событий

Здесь практически нет отличий от работы с сервисом **Pusher**. Однако взгляните на пример 16.28, который демонстрирует, как с помощью **Echo** можно прослушивать публичные каналы в случае простейшей трансляции событий.

Пример 16.28. Прослушивание публичного канала с помощью **Echo**

```
var currentTeamId = 5; // Обычно задается в другом месте
```

```
Echo.channel(`teams.${currentTeamId}`)
  .listen('UserSubscribed', (data) => {
    console.log(data);
  });
```

Пакет **Echo** предлагает несколько методов для подписки на каналы различных типов. Метод `channel()` осуществляет подписку на публичный канал. При прослушивании события с помощью **Echo** можно не указывать полное пространство имен события; достаточно указать только его уникальное имя класса.

После этого у нас будет доступ к передаваемым вместе с событием публичным данным, представленным в виде объекта `data`. Мы также можем создать цепочку из нескольких методов-обработчиков `listen()`, как показано в примере 16.29.

Пример 16.29. Создание цепочки слушателей событий с помощью **Echo**

```
Echo.channel(`teams.${currentTeamId}`)
  .listen('UserSubscribed', (data) => {
    console.log(data);
  })
  .listen('UserCanceled', (data) => {
    console.log(data);
  });
```



Не забывайте компилировать и включать код!

Вы попробовали запустить эти примеры кода и не увидели никаких изменений в браузере? Тогда посмотрите, не забыли ли вы выполнить команду `npm run dev` (в случае однократного запуска) или `npm run watch` (при запуске слушателя) для компиляции кода. Если вы еще этого не сделали, подключите файл `app.js` где-нибудь в своем шаблоне.

Приватные каналы и простейшая аутентификация

Пакет Echo также предлагает метод для подписки на приватные каналы: `private()`. Он работает аналогично методу `channel()`, но требует настройки определения авторизации канала в файле `routes/channel.php`, как было описано ранее. В отличие от использования SDK, перед именем канала не нужно ставить префикс `private-`.

В примере 16.30 показано, как обеспечить прослушивание приватного канала с именем `private-teams.5`.

Пример 16.30. Прослушивание приватного канала с помощью Echo

`var currentTeamId = 5; // Обычно задается в другом месте`

```
Echo.private(`teams.${currentTeamId}`)
  .listen('UserSubscribed', (data) => {
    console.log(data);
  });
```

Каналы присутствия

Пакет Echo существенно упрощает задачу присоединения к каналу присутствия и прослушивания событий в них. На этот раз нужно выполнить привязку к каналу с помощью метода `join()`, как в примере 16.31.

Пример 16.31. Присоединение к каналу присутствия

`var currentTeamId = 5; // Обычно задается в другом месте`

```
Echo.join(`teams.${currentTeamId}`)
  .here((members) => {
    console.log(members);
  });
```

Метод `join()` осуществляет подписку на канал присутствия, а `here()` позволяет определить поведение для случая присоединения к каналу присутствия данного пользователя, а также присоединения/отсоединения любых других пользователей.

Канал присутствия представляет собой что-то вроде боковой панели в комнате чата, показывающей, какие пользователи в сети. Когда вы в первый раз присоединяетесь к каналу присутствия, производится обратный вызов метода `here()` с предоставлением ему текущего списка всех членов. Затем при каждом присоединении/отсоединении кого-либо из пользователей этот обратный вызов повторяется с предоставлением ему обновленного списка. При этом не выводятся какие-либо сообщения, но можно подавать звуковой сигнал, обновлять список пользователей на странице или как-то иначе реагировать.

Есть специальные методы для каждого вида событий, которые можно использовать по отдельности или в составе цепочки методов (пример 16.32).

Пример 16.32. Прослушивание разных видов событий присутствия

`var currentTeamId = 5; // Обычно задается в другом месте`

```
Echo.join('teams.' + currentTeamId)
  .here((members) => {
    // Запускаем, когда подключаемся сами
    console.table(members);
  })
  .joining((joiningMember, members) => {
    // Запускаем, когда подключается другой пользователь
    console.table(joiningMember);
  })
  .leaving((leavingMember, members) => {
    // Запускаем, когда отключается другой пользователь
    console.table(leavingMember);
  });
```

Исключение текущего пользователя

Ранее уже говорилось, что исключить текущего пользователя из числа его получателей можно вызовом глобального хелпера `broadcast()` вместо глобального хелпера `event()`, с пристыковкой к нему метода `toOthers()`. Однако в случае Echo данная задача будет решена с помощью JavaScript-кода. Все заработает без каких-либо усилий с вашей стороны.

JavaScript-библиотека Echo не делает ничего, что вы не могли бы сделать сами. Но существенно упрощает выполнение многих распространенных задач, предоставляя более чистый и выразительный синтаксис для типичных задач использования веб-сокетов.

Подписка на уведомления с помощью Echo

Механизм уведомлений Laravel, предлагаемый по умолчанию, поддерживает драйвер трансляции, который выдает уведомления как транслируемые события. Чтобы подписаться на них с помощью Echo, воспользуйтесь методом `Echo.notification()` (пример 16.33).

Пример 16.33. Подписка на уведомление с помощью Echo

```
Echo.private(`App.User.${userId}`)
  .notification((notification) => {
    console.log(notification.type);
  });
```

События клиента

Если нужен обмен быстрыми высокопроизводительными сообщениями между вашими пользователями без непосредственного направления этих сообщений в приложение Laravel — например, для уведомления участников чата, что другой пользователь набирает текст, — воспользуйтесь методом `whisper()` пакета Echo, как в примере 16.34.

Пример 16.34. Отправка сообщений в обход сервера Laravel с помощью метода `whisper()` пакета Echo

```
Echo.private('room')
    .whisper('typing', {
        name: this.user.name
    });
```

После этого нужно обеспечить прослушивание с помощью метода `listenForWhisper()`, как показано в примере 16.35.

Пример 16.35. Прослушивание событий клиента с помощью Echo

```
Echo.private('room')
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

Планировщик

Если вам приходилось писать задания `cron`, то наверняка появилось желание сменить этот инструмент на что-то получше. Помимо того что вам приходится иметь дело с неудобным и плохо запоминающимся синтаксисом, вы также не можете сохранить этот важный аспект приложения в системе управления версиями.

Планировщик Laravel позволяет существенно упростить работу с запланированными задачами. Определите их в коде, а затем направьте одно задание `cron` на ваше приложение, выполняя раз в минуту команду `php artisan schedule:run`. При каждом запуске этой команды Artisan фреймворк будет сверяться с вашими определениями графиков, не пришло ли время выполнения какой-либо запланированной задачи.

Вот как выглядит задание `cron` для определения этой команды:

```
* * * * * cd /home/myapp.com && php artisan schedule:run >> /dev/null 2>&1
```

При планировании можно использовать много временных интервалов и типов задач.

Класс `app/Console/Kernel.php` обладает методом `schedule()`. Именно здесь следует определять те задачи, выполнение которых вам нужно запланировать.

Доступные типы задач

Для начала рассмотрим предельно простой вариант. Допустим, нам нужно выполнять замыкание с интервалом 1 минута (пример 16.36). Каждый раз, когда выполнение задания стоп дойдет до команды `schedule:run`, будет вызываться это замыкание.

Пример 16.36. Планирование выполнения замыкания с интервалом 1 минута

```
// app/Console/Kernel.php
public function schedule(Schedule $schedule)
{
    $schedule->call(function () {
        CalculateTotals::dispatch();
    })->everyMinute();
}
```

При планировании можно использовать еще два типа задач: команды Artisan и командной оболочки.

Вы можете запланировать команду Artisan, передав ее синтаксис в том же виде, как она выглядит при вызове из командной строки:

```
$schedule->command('scores:tally --reset-cache')->everyMinute();
```

Или запланировать любые команды командной оболочки, которые можно выполнить с помощью метода `exec()` языка PHP:

```
$schedule->exec('/home/myapp.com/bin/build.sh')->everyMinute();
```

Доступные временные интервалы

Планировщик позволяет не только определять в коде свои задачи, но и планировать их там же. Laravel дает возможность отслеживать время и проверять, когда выполнять ту или иную задачу. Это можно сделать с помощью метода `everyMinute()`, потому что ответом всегда является запуск задачи. Однако Laravel делает предельно простым и остальные составляющие планирования задач, даже в самых сложных случаях.

Рассмотрим доступные варианты из очень сложного случая, который тем не менее легко определяется в Laravel:

```
$schedule->call(function () {
    // Еженедельный запуск по воскресеньям в 23:50
})->weekly()->sundays()->at('23:50');
```

Здесь составляется цепочка из различных определений времени: сначала указывается периодичность запуска, затем день недели и время суток, и это еще не все, что можно сделать.

В табл. 16.1 представлен список модификаторов даты/времени, доступных при планировании заданий.

Таблица 16.1. Модификаторы даты/времени, доступные при использовании планировщика

Команда	Описание
->timezone('America/Detroit')	Установка часового пояса для графиков
->cron('* * * * *')	Определение графика с использованием традиционной нотации cron
->everyMinute()	Запуск с периодичностью 1 минута
->everyFiveMinutes()	Запуск с периодичностью 5 минут
->everyTenMinutes()	Запуск с периодичностью 10 минут
->everyThirtyMinutes()	Запуск с периодичностью 30 минут
->hourly()	Запуск с периодичностью 1 час
->daily()	Ежедневный запуск в полночь
->dailyAt('14:00')	Ежедневный запуск в 14:00
->twiceDaily(1, 14)	Ежедневный запуск в 1:00 и 14:00
->weekly()	Еженедельный запуск (в полночь по воскресеньям)
->weeklyOn(5, '10:00')	Еженедельный запуск по пятницам в 10:00
->monthly()	Ежемесячный запуск (в полночь в первый день месяца)
->monthlyOn(15, '23:00')	Ежемесячный запуск в 15-й день месяца в 23:00
->quarterly()	Запуск раз в квартал (в полночь 1 января, апреля, июля и октября)
->yearly()	Запуск раз в год (в полночь 1 января)
->when(closure)	Запуск только в случае, если замыкание возвращает значение true
->skip(closure)	Запуск только в случае, если замыкание возвращает значение false
->between('8:00', '12:00')	Запуск только в промежутке между указанными моментами времени
->unlessBetween('8:00', '12:00')	Запуск в любое время, за исключением промежутка между указанными моментами времени
->weekdays()	Запуск только в рабочие дни
->sundays()	Запуск только по воскресеньям
->mondays()	Запуск только по понедельникам
->tuesdays()	Запуск только по вторникам
->wednesdays()	Запуск только по средам
->thursdays()	Запуск только по четвергам
->fridays()	Запуск только по пятницам
->saturdays()	Запуск только по субботам

Почти все эти методы можно включать в цепочку вызовов, кроме случаев, когда получаемая комбинация не имеет смысла.

Ряд возможных комбинаций показан в примере 16.37.

Пример 16.37. Некоторые примеры запланированных событий

```
// В обоих случаях еженедельный запуск по воскресеньям в 23:50
$schedule->command('do:thing')->weeklyOn(0, '23:50');
$schedule->command('do:thing')->weekly()->sundays()->at('23:50');

// Запуск с периодичностью 1 час по рабочим дням с 8 утра до 5 вечера
$schedule->command('do:thing')->weekdays()->hourly()->when(function () {
    return date('H') >= 8 && date('H') <= 17;
});

// Запуск с периодичностью 1 час по рабочим дням с 8 утра до 5 вечера
// с использованием метода between
$schedule->command('do:thing')->weekdays()->hourly()->between('8:00', '17:00');

// Запуск с периодичностью 30 минут, за исключением того случая, когда
// SkipDetector отменяет запуск
$schedule->command('do:thing')->everyThirtyMinutes()->skip(function () {
    return app('SkipDetector')->shouldSkip();
});
```

Определение часовых поясов для запланированных задач

Вы можете определить часовой пояс для конкретной запланированной задачи, используя метод `timezone()`:

```
$schedule->command('do:it')->weeklyOn(0, '23:50')->timezone('America/Chicago');
```

С версии 5.8 можно задать часовой пояс по умолчанию (независимый от часового пояса приложения), который будет применяться для отсчета интервалов всех запланированных задач. Нужно определить метод `scheduleTimezone()` в классе `App\Console\Kernel`:

```
protected function scheduleTimezone()
{
    return 'America/Chicago';
}
```

Блокирование и наложение

Чтобы исключить возможность наложения задач друг на друга (например, если выполнение задачи, запускаемой с периодичностью 60 секунд, иногда занимает больше минуты), поставьте в конце цепочки вызовов метод `withoutOverlapping()`.

Этот метод отменяет запуск задачи, если еще выполняется ее предыдущий экземпляр:

```
$schedule->command('do:thing')->everyMinute()->withoutOverlapping();
```

Обработка выходных данных задачи

Иногда нужно получить выходные данные запланированной задачи — возможно, для целей логирования, отправки уведомлений или чтобы убедиться в успешном выполнении задачи.

Для записи возвращаемых задачами данных в файл воспользуйтесь методом `sendOutputTo()`:

```
$schedule->command('do:thing')->daily()->sendOutputTo($filePath);
```

Если вместо этого нужно добавить данные в уже имеющийся файл, примените метод `appendOutputTo()`:

```
$schedule->command('do:thing')->daily()->appendOutputTo($filePath);
```

А если выходные данные надо отослать на конкретный адрес электронной почты, то запишите их в файл, после чего вызовите метод `emailOutputTo()`:

```
$schedule->command('do:thing')
->daily()
->sendOutputTo($filePath)
->emailOutputTo('me@myapp.com');
```

Проследите, чтобы в базовой конфигурации электронной почты Laravel были заданы правильные параметры электронной почты.



Задания, запланированные с помощью замыканий, не могут отправлять выходные данные

Методы `sendOutputTo()`, `appendOutputTo()` и `emailOutputTo()` работают только тогда, когда задача запланирована с помощью метода `command()`. К сожалению, их нельзя использовать для замыканий.

Часто требуется отправить выходные данные веб-хуку, чтобы убедиться в надлежащем выполнении задачи. Такой контроль работоспособности предоставляют несколько сервисов. Наиболее значительные из них: **Laravel Envoyer** (<https://envoyer.io>) — сервис для развертывания с нулевым временем простоя, который также обеспечивает контроль работоспособности cron, и **Dead Man's Snitch** (<https://deadmanssnitch.com/>) — инструмент исключительно для контроля работоспособности заданий cron.

Эти услуги не требуют отправки им каких-либо электронных писем. Вместо этого им нужно отсылать проверочный HTTP-«пинг». Для этого в Laravel есть методы `pingBefore()` и `thenPing()`:

```
$schedule->command('do:thing')
    ->daily()
    ->pingBefore($beforeUrl)
    ->thenPing($afterUrl);
```

Для их использования нужно подгрузить пакет Guzzle с помощью утилиты Composer.

```
composer require guzzlehttp/guzzle
```

Перехват задач

Запустить что-то *до* или *после* определенной задачи можно с помощью методов-хуков `before()` и `after()`:

```
$schedule->command('do_thing')
    ->daily()
    ->before(function () {
        // Подготовка
    })
    ->after(function () {
        // Очистка
    });
```

Тестирование

Тестировать очереди заданий (либо любой другой очереди) легко. В файле `phpunit.xml`, содержащем используемые для тестов параметры конфигурации, переменной среды `QUEUE_DRIVER` (драйвер очереди) по умолчанию присваивается значение `sync`. Так тесты будут обрабатывать очередь заданий (и любую другую очередь) синхронно, непосредственно в коде, без какой-либо системы очередей. То есть можно проверять задания точно так же, как любой другой код.

Если нужно убедиться в успешном запуске задания, то поможет метод `expectsJobs()`, как показано в примере 16.38.

Пример 16.38. Проверка на предмет того, было ли отправлено задание указанного класса

```
public function test_changing_number_of_subscriptions_crunches_reports()
{
    $this->expectsJobs(\App\Jobs\CrunchReports::class);

    ...
}
```

5.3 В проектах, использующих Laravel 5.3 или более новую версию, можно выполнять непосредственную проверку конкретного задания (пример 16.39).

Пример 16.39. Использование замыкания для проверки отправленного задания на предмет соответствия заданным критериям

```
use Illuminate\Support\Facades\Bus;
...
public function test_changing_subscriptions_triggers_crunch_job()
{
    ...
    Bus::fake();

    Bus::assertDispatched(CrunchReports::class, function ($job) {
        return $job->subscriptions->contains(5);
    });

    // Также можно использовать метод assertNotDispatched()
}
```

Убедиться в успешном запуске события можно тремя способами. Во-первых, проверить, было ли выполнено ожидаемое поведение, не обращая внимания на само событие.

5.2 Во-вторых, с версии 5.2 можно непосредственно проверить, было ли запущено событие, как показано в примере 16.40.

Пример 16.40. Проверка на предмет того, было ли запущено событие указанного класса

```
public function test_usersubscribed_event_fires()
{
    $this->expectsEvents(\App\Events\UserSubscribed::class);

    ...
}
```

5.3 В-третьих, с версии 5.3 можно тестировать запущенное событие, как показано в примере 16.41.

Пример 16.41. Использование замыкания для проверки запущенного события на предмет соответствия заданным критериям

```
use Illuminate\Support\Facades\Event;
...
public function test_usersubscribed_event_fires()
{
    Event::fake();

    ...

    Event::assertDispatched(UserSubscribed::class, function ($e) {
        return $e->user->email = 'user-who-subscribed@mail.com';
    });

    // Также можно использовать метод assertNotDispatched()
}
```

Тестируемый код периодически запускает события, и вам нужно отключить слушатели событий на время проверки. Отключить систему событий можно методом `withoutEvents()`, как показано в примере 16.42.

Пример 16.42. Отключение слушателей событий на время выполнения теста

```
public function test_something_subscription_related()
{
    $this->withoutEvents();

    ...
}
```

Резюме

Очереди позволяют перенести ряд фрагментов кода вашего приложения из синхронной последовательности взаимодействия с пользователями в отдельный список команд, обрабатываемый обработчиком очередей. Это позволяет не прерывать взаимодействие приложения с пользователями, выполняя обработку более медленных процессов асинхронно в фоновом режиме.

Задания — это классы, структура которых позволяет инкапсулировать фрагменты поведения приложения так, чтобы их можно было помещать в очередь.

Система событий Laravel соответствует шаблону публикации/подписки, иначе говоря, шаблону наблюдателя. Можно отправлять уведомления о событиях из одной части приложения, разместив где-то в другом месте слушатели этих уведомлений, указывающие, какое поведение должно выполняться в ответ на то или иное событие. Веб-сокетами транслируются события в клиентские интерфейсы.

Планировщик Laravel упрощает планирование задач. Направьте ежеминутное задание `stop` на команду `php artisan schedule:run`, а затем запланируйте свои задачи с помощью планировщика, при необходимости используя самые сложные временные настройки. Фреймворк выполнит за вас всю работу по отсчитыванию этих интервалов.

17

Хелперы и коллекции

Мы уже рассмотрели много глобальных хелперов (helpers), призванных облегчить выполнение распространенных задач, включая `dispatch()` для отправки заданий, `event()` для запуска событий и `app()` — для разрешения зависимостей. В главе 5 мы немного коснулись темы коллекций Laravel — этих «массивов на стероидах».

В данной главе рассмотрим ряд наиболее часто используемых, мощных хелперов и некоторые базовые приемы программирования с использованием коллекций.

Хелперы

С полным списком вспомогательных функций Laravel можно ознакомиться в его документации по адресу <http://bit.ly/2HQAFC>. Здесь рассмотрим лишь ряд наиболее полезных функций.



 В версии 5.8 все глобальные хелперы с префиксом `array_` или `str_` были отнесены к числу нерекомендуемых. В версии 5.9 эти вспомогательные функции будут удалены, но оставлены доступными в виде отдельного пакета для обеспечения обратной совместимости. За каждым из них стоит некоторый метод фасада `Arr` или `Str`, поэтому можно подготовиться к этому будущему, используя методы фасадов или запланировав загрузку дополнительного пакета после того, как он станет доступным.

Массивы

Хотя язык PHP предлагает достаточно мощные встроенные функции для работы с массивами, некоторые виды стандартных манипуляций все же требуют громоздких циклов и проверочной логики. Хелперы для работы с массивами фреймворка

Laravel существенно упрощают выполнение над массивами ряда распространенных манипуляций.

- ❑ `array_first($array, $callback, $default = null)`. Возвращает первое значение массива, которое удовлетворяет условию, определенному в замыкании обратного вызова. В качестве необязательного третьего параметра можно задать значение по умолчанию. Вот пример:

```
$people = [
    [
        'email' => 'm@me.com',
        'name' => 'Malcolm Me'
    ],
    [
        'email' => 'j@jo.com',
        'name' => 'James Jo'
    ],
];

$value = array_first($people, function ($person, $key) {
    return $person['email'] == 'j@jo.com';
});
```

- ❑ `array_get($array, $key, $default = null)`. Позволяет легко извлекать значения из массива, предоставляя два дополнительных преимущества: не выдает сообщение об ошибке, когда вы запрашиваете несуществующий ключ (позволяя предоставить значение по умолчанию в качестве третьего параметра), и дает использовать точечную нотацию для обхода вложенных массивов. Например:

```
$array = ['owner' => ['address' => ['line1' => '123 Main St.']]];

$line1 = array_get($array, 'owner.address.line1', 'No address');
$line2 = array_get($array, 'owner.address.line2');
```

- ❑ `array_has($array, $keys)`. Позволяет проверить, задано ли конкретное значение массива, используя точечную нотацию для обхода вложенных массивов. Параметр `$keys` может быть как единичным элементом, так и массивом элементов, тогда проверяется наличие в массиве каждого из этих элементов:

```
$array = ['owner' => ['address' => ['line1' => '123 Main St.']]];

if (array_has($array, 'owner.address.line2')) {
    // Выполнение определенных действий
}
```

- ❑ `array_pluck($array, $value, $key = null)`. Возвращает массив значений, соответствующих предоставленному ключу:

```
$array = [
    ['owner' => ['id' => 4, 'name' => 'Tricia']],
```

```
    ['owner' => ['id' => 7, 'name' => 'Kimberly']],
];
```

```
$array = array_pluck($array, 'owner.name');
```

```
// Возвращает ['Tricia', 'Kimberly'];
```

Если нужно, чтобы в возвращаемом массиве в качестве ключа использовалось другое значение исходного массива, то передайте как третий параметр ссылку на это значение, применяя точечную нотацию:

```
$array = array_pluck($array, 'owner.name', 'owner.id');
```

```
// Возвращает [4 => 'Tricia', 7 => 'Kimberly'];
```

- ❑ `array_random($array, $num = null)`. Возвращает случайный элемент из предоставленного массива. В случае предоставления параметра `$num` возвращает массив, содержащий соответствующее число случайно отобранных элементов:

```
$array = [
    ['owner' => ['id' => 4, 'name' => 'Tricia']],
    ['owner' => ['id' => 7, 'name' => 'Kimberly']],
];
```

```
$randomOwner = array_random($array);
```

Строки

Язык PHP предлагает некоторые встроенные функции для работы со строками и выполнения в отношении них проверок, использование которых часто приобретает слишком громоздкий характер. Хелперы Laravel позволяют быстрее и проще выполнять ряд распространенных строковых операций.

- ❑ `e($string)`. Псевдоним для метода `htmlentities()`. Подготавливает строку (часто предоставленную пользователем) для безопасного вывода на HTML-странице. Например:

```
e('<script>do something nefarious</script>');
```

```
// Возвращает &lt;script&gt;do something nefarious&lt;/script&gt;
```

- ❑ `starts_with($haystack, $needle)`, `ends_with($haystack, $needle)` и `str_contains($haystack, $needle)`. Возвращают логическое значение, указывающее, начинается ли строка `$haystack` со строки `$needle`, заканчивается ли она этой строкой и есть ли в ней вообще эта строка:

```
if (starts_with($url, 'https')) {
    // Выполнение некоторых действий
}
```

```
if (ends_with($abstract, '...')) {
```

```

    // Выполнение некоторых действий
}

if (str_contains($description, '1337 h4x0r')) {
    // Выполнение некоторых действий
}

```

- ❑ **str_limit(\$value, \$limit = 100, \$end = '...')**. Ограничивает длину строки указанным количеством символов. Если длина строки меньше указанного предела, просто возвращает строку. Если же она превышает предел, обрезает строку до указанного количества символов, добавляя в конце ... или предоставленную строку \$end. Например:

```

$abstract = str_limit($loremIpsum, 30);

// Возвращает "Lorem ipsum dolor sit amet, co..."

$abstract = str_limit($loremIpsum, 30, "&hellip;");

// Возвращает "Lorem ipsum dolor sit amet, co&hellip;";

```

- ❑ **str_is(\$pattern, \$value)**. Возвращает логическое значение, указывающее, соответствует ли данная строка заданному шаблону. В качестве шаблона используется регулярное выражение, в котором символы звездочки обозначают любое количество произвольных символов:

```

str_is('*.dev', 'myapp.dev'); // true
str_is('*.dev', 'myapp.dev.co.uk'); // false
str_is('*dev*', 'myapp.dev'); // true
str_is('*myapp*', 'www.myapp.dev'); // true
str_is('my*app', 'myfantasticapp'); // true
str_is('my*app', 'myapp'); // true

```



Как передать регулярное выражение методу str_is()

Если интересно, какие регулярные выражения можно передавать методу str_is(), взгляните на представленное здесь определение этой функции (я немного его сократил для экономии пространства). Обратите внимание, что это псевдоним для метода Illuminate\Support\Str::is():

```

public function is($pattern, $value)
{
    if ($pattern == $value) return true;

    $pattern = preg_quote($pattern, '#');
    $pattern = str_replace('\*', '.*', $pattern);
    if (preg_match('#^'.$pattern.'\z#u', $value) === 1) {
        return true;
    }

    return false;
}

```


- ❑ `str_random($length = n)`. Возвращает случайную последовательность из указанного числа буквенно-цифровых символов как верхнего, так и нижнего регистра:

```
$hash = str_random(64);
```

```
// Пример результата:
```

```
// J40uNwAvY60wE4BPEWxu7BZFQEmxEHmGiLmQncj0ThMGJK705Kfgptyb9ulwspmh
```

- ❑ `str_slug($title, $separator = '-', $language = 'en')`. Возвращает URL-совместимый слаг на основе строки — часто используется при создании сегмента URL-адреса для названия/заголовка:

```
str_slug('How to Win Friends and Influence People');
```

```
// Возвращает 'how-to-win-friends-and-influence-people'
```

- ❑ `str_plural($value, $count = n)`. Преобразует строку в форму множественного числа. На данный момент поддерживает только английский язык:

```
str_plural('book');
```

```
// Возвращает books
```

```
str_plural('person');
```

```
// Возвращает people
```

```
str_plural('person', 1);
```

```
// Возвращает person
```

- ❑ `__($key, $replace = [], $locale = null)`. Осуществляет перевод предоставленной строки/ключа с использованием ваших файлов локализации:

```
echo __('Welcome to your dashboard');
```

```
echo __('messages.welcome');
```

Пути приложения

При работе с файловой системой часто приходится тратить много усилий на создание ссылок на определенные каталоги для извлечения и сохранения файлов. Перечисленные здесь хелперы позволяют быстро получить полный путь к некоторым наиболее важным каталогам вашего приложения.

Все эти хелперы могут вызываться без параметров, но если передается параметр, он добавляется к обычной строке каталога, после чего возвращается единый результат.

- ❑ `app_path($append = '')`. Возвращает путь к каталогу `app`:

```
app_path();
```

```
// Возвращает /home/forgem/app.com/app
```

- ❑ `base_path($path = '')`. Возвращает путь к корневому каталогу вашего приложения:

```
base_path();

// Возвращает /home/forge/myapp.com
```

- ❑ `config_path($path = '')`. Возвращает путь к файлам конфигурации вашего приложения:

```
config_path();

// Возвращает /home/forge/myapp.com/config
```

- ❑ `database_path($path = '')`. Возвращает путь к файлам базы данных вашего приложения:

```
database_path();

// Возвращает /home/forge/myapp.com/database
```

- ❑ `storage_path($path = '')`. Возвращает путь к каталогу хранения **storage** вашего приложения:

```
storage_path();

// Возвращает /home/forge/myapp.com/storage
```

URL-адреса

Иногда пути к определенным файлам клиентского интерфейса — например, к ресурсам — являются постоянными, но их набор может отнимать много сил. В таком случае полезны удобные псевдонимы этих путей, которые мы рассмотрим в этом разделе. Иногда пути могут изменяться при изменении определений маршрутов или версировании новых файлов с помощью Mix, так что некоторые из этих вспомогательных функций играют важную роль в обеспечении надлежащей работы ссылок и ресурсов.

- ❑ `action($action, $parameters = [], $absolute = true)`. При условии, что методу контроллера поставлен в соответствие отдельный URL-адрес, возвращает корректный URL-адрес для предоставленной пары имен контроллера и метода, которые разделяются символом @ или записываются в нотации кортежей:

```
<a href="{{ action('PeopleController@index') }}">See all People</a>
// Или с использованием нотации кортежей:
<a href=
    "{{ action([App\Http\Controllers\PeopleController::class, 'index']) }}">
    See all People
</a>

// Возвращает <a href="http://myapp.com/people">See all People</a>
```

Если метод контроллера требует параметры, можно передать их в качестве второго параметра (в виде массива, если метод принимает несколько обязательных параметров). Для ясности их можно снабдить ключами. Главное, чтобы они были расположены в правильном порядке:

```
<a href="{{ action('PeopleController@show', ['id' => 3]) }}">See Person #3</a>
// или
<a href="{{ action('PeopleController@show', [3]) }}">See Person #3</a>

// Возвращает <a href="http://myapp.com/people/3">See Person #3</a>
```

Если вы передадите значение `false` в качестве третьего параметра, то будет сгенерирована относительная (`/people/3`), а не абсолютная ссылка (`http://myapp.com/people/3`).

- ❑ `route($name, $parameters = [], $absolute = true)`. Возвращает URL-адрес маршрута с указанным именем:

```
// routes/web.php
Route::get('people', 'PeopleController@index')->name('people.index');

// Где-нибудь в представлении
<a href="{{ route('people.index') }}">See all People</a>

// Возвращает <a href="http://myapp.com/people">See all People</a>
```

Если определение маршрута требует параметры, можно передать их в качестве второго параметра (в виде массива, если требуется несколько параметров). Для ясности их можно снабдить ключами. Главное, чтобы они были расположены в правильном порядке:

```
<a href="{{ route('people.show', ['id' => 3]) }}">See Person #3</a>
// или
<a href="{{ route('people.show', [3]) }}">See Person #3</a>

// Возвращает <a href="http://myapp.com/people/3">See Person #3</a>
```

Если вы передадите значение `false` в качестве третьего параметра, то будет сгенерирована относительная, а не абсолютная ссылка.

- ❑ `url($string)` и `secure_url($string)`. Преобразует любую предоставленную строку пути в полностью квалифицированный URL-адрес (`secure_url()` делает то же самое, что и `url()`, но использует протокол HTTPS):

```
url('people/3');

// Возвращает http://myapp.com/people/3
```

При вызове без параметров возвращает экземпляр класса `Illuminate\Routing\UrlGenerator`, что позволяет пристыковать метод:

```
url()->current();
// Возвращает http://myapp.com/abc
```

```
url()->full();
// Возвращает http://myapp.com/abc?order=reverse

url()->previous();
// Возвращает http://myapp.com/login
// И это еще далеко не все методы, доступные в классе UrlGenerator...
```

- ❑ `mix($path, $manifestDirectory = '')`. Если для ресурсов задаются версии с помощью системы управления версиями Mix, возвращает полностью квалифицированный URL-адрес версированного файла для предоставленного неверсированного имени пути:

```
<link rel="stylesheet" href="{ mix('css/app.css') }">

// Возвращает что-то наподобие /build/css/app-eb555e38.css
```



Использование хелпера `elixir()` до версии 5.4

В проектах, использующих версии до Laravel 5.4, вместо хелпера `mix()` следует использовать `elixir()`. Более подробные сведения см. в документации по адресу <http://bit.ly/2ACcHu1>.

Прочее

Существует и ряд других глобальных хелперов, с которыми стоит ознакомиться. Рекомендую изучить весь список (<http://bit.ly/2H9KaFC>), но перечислю здесь те, которые определенно заслуживают вашего внимания.

- ❑ `abort($code, $message, $headers)`, `abort_unless($boolean, $code, $message, $headers)` и `abort_if($boolean, $code, $message, $headers)`. Выбрасывают HTTP-исключения. Метод `abort()` выбрасывает определенное вами исключение, `abort_unless()` выбрасывает его, если первый параметр равен `false`. Метод `abort_if()` выбрасывает его, если первый параметр равен `true`:

```
public function controllerMethod(Request $request)
{
    abort(403, 'You shall not pass');
    abort_unless(request()->filled('magicToken'), 403);
    abort_if(request()->user()->isBanned, 403);
}
```

- ❑ `auth()`. Возвращает экземпляр аутентификатора Laravel. Подобно фасаду `Auth`, его можно использовать для получения текущего пользователя, проверки статуса аутентификации и т. д.:

```
$user = auth()->user();
$userId = auth()->id();

if (auth()->check()) {
    // Выполнение некоторых действий
}
```

- ❑ **back()**. Формирует ответ «перенаправления назад», отправляя пользователя в предыдущее место:

```
Route::get('post', function () {
    ...

    if ($condition) {
        return back();
    }
});
```

- ❑ **collect(\$array)**. Принимает массив и возвращает те же данные, преобразованные в коллекцию:

```
$collection = collect(['Rachel', 'Hototo']);
```

Мы подробно рассмотрим коллекции в следующем разделе.

- ❑ **config(\$key)**. Возвращает значение для любого элемента конфигурации, указанного с помощью точечной нотации:

```
$defaultDbConnection = config('database.default');
```

- ❑ **csrf_field()** и **csrf_token()**. Возвращают полный HTML-код скрытого ввода (**csrf_field()**) или только значение соответствующего токена (**csrf_token()**) для добавления CSRF-проверки при отправке формы:

```
<form>
    {{ csrf_field() }}
</form>
```

// или

```
<form>
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

- ❑ **dd(\$variable...)**. Сокращение от слов dump and die — «выгрузить и умереть». Выполняет метод **var_dump()** для всех указанных параметров, а затем — метод **exit()** для завершения работы приложения (используется для целей отладки):

```
...
dd($var1, $var2, $state); // Почему это не работает ???
```

- ❑ **env(\$key, \$default = null)**. Возвращает значение переменной среды для указанного ключа:

```
$key = env('API_KEY', '');
```

Помните о том, что метод **env()** не следует использовать за пределами конфигурационных файлов.

- ❑ **dispatch(\$job)**. Отправляет задание:

```
dispatch(new EmailAdminAboutNewUser($user));
```

- ❑ `event($event)`. Запускает событие:

```
event(new ContactAdded($contact));
```

- ❑ `factory($entityClass)`. Возвращает экземпляр генератора фабрики для указанного класса:

```
$contact = factory(App\Contact::class)->make();
```

- ❑ `old($key = null, $default = null)`. Возвращает старое значение (из формы, отправленной пользователем в прошлый раз) для указанного ключа формы, если он существует:

```
<input name="name" value="{{ old('value', 'Your name here') }}"
```

- ❑ `redirect($path)`. Возвращает ответ перенаправления на указанный путь:

```
Route::get('post', function () {
    ...

    return redirect('home');
});
```

При вызове без параметров генерирует экземпляр класса `Illuminate\Routing\Redirector`.

- ❑ `response($content, $status = 200, $headers)`. При вызове с параметрами возвращает предварительно собранный экземпляр класса `Response`. При вызове без параметров возвращает экземпляр фабрики класса `Response`:

```
return response('OK', 200, ['X-Header-Greatness' => 'Super great']);

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

- ❑ `view($viewPath)`. Возвращает экземпляр представления:

```
Route::get('home', function () {
    return view('home'); // Возвращает /resources/views/home.blade.php
});
```

Коллекции

Коллекции — один из самых мощных инструментов Laravel, который пока не получил достаточного признания. Мы уже немного говорили о них в подразделе «Коллекции Eloquent» на с. 154, но будет не лишним кратко вспомнить, что они собой представляют.

Коллекции — это массивы со «сверхспособностями». Все те методы для обхода массивов, которые обычно принимают массив (`array_walk()`, `array_map()`, `array_reduce()` и т. д.) и сбивают с толку непостоянством в плане сигнатуры метода, доступны в каждой коллекции в виде единообразных, чистых, стыкуемых методов.

Коллекции позволяют вам наслаждаться функциональным программированием и получать более чистый код за счет отображения, свертки и фильтрации.

Приведу здесь некоторые базовые сведения о коллекциях Laravel и программировании конвейера коллекций, но для более глубокого изучения рекомендую прочитать книгу Адама Уотана *Refactoring to Collections* (Gumroad).

Базовые сведения

Коллекции не уникальное нововведение Laravel. Многие языки программирования «из коробки» предлагают аналогичные возможности для работы с массивами. Однако нам не очень повезло с этим в случае языка PHP.

Используя функции `array*()` языка PHP, мы можем взять такой громоздкий код, какой показан в примере 17.1, и преобразовать его к чуть менее громоздкому виду, как в примере 17.2.

Пример 17.1. Часто используемый, но совершенно незлегантный цикл `foreach`

```
$users = [...];

$admins = [];

foreach ($users as $user) {
    if ($user['status'] == 'admin') {
        $user['name'] = $user['first'] . ' ' . $user['last'];
        $admins[] = $user;
    }
}

return $admins;
```

Пример 17.2. Рефакторинг цикла `foreach` с использованием встроенных функций языка PHP

```
$users = [...];

return array_map(function ($user) {
    $user['name'] = $user['first'] . ' ' . $user['last'];
    return $user;
}, array_filter($users, function ($user) {

    return $user['status'] == 'admin';
}));
```

Здесь мы избавились от временной переменной (`$admins`) и преобразовали один сложный цикл `foreach` в две отдельные операции отображения и фильтрации.

Проблема в том, что функции для работы с массивами языка PHP чрезвычайно неудобны и сложны в использовании. Например, в данном случае метод `array_map()` принимает замыкание в качестве первого параметра и массив как второй параметр. Метод `array_filter()`, наоборот, принимает массив в качестве первого параметра

и замыкание вторым параметром. Кроме того, если мы возьмем чуть более сложный код, то в результате получим сплошной беспорядок, где одни функции будут обертывать другие функции, которые, в свою очередь, будут обертывать третьи функции.

Коллекции Laravel берут те мощные возможности, которыми обладают методы для работы с массивами языка PHP, и снабжают их чистым текущим синтаксисом, добавляя к этому множество методов, которых нет в арсенале методов для работы с массивами языка PHP. Используя хелпер `collect()`, который преобразует массив в коллекцию Laravel, можно получить код, показанный в примере 17.3.

Пример 17.3. Рефакторинг цикла `foreach` с использованием коллекций Laravel

```
$users = collect([...]);

return $users->filter(function ($user) {
    return $user['status'] == 'admin';
})->map(function ($user) {
    $user['name'] = $user['first'] . ' ' . $user['last'];
    return $user;
});
```

Это далеко не самый яркий пример. Во множестве случаев применение коллекций обеспечивает намного более значительное сокращение и упрощение кода. Однако надо сказать, что данный пример *очень типичен*.

Взгляните на код исходного примера и оцените, насколько он грязный. Вы не можете сказать, для чего нужен тот или иной фрагмент, пока не поймете назначение всего этого кода в целом.

Самое большое преимущество коллекций в том, что можно представить выполняемую над массивом операцию в виде ряда простых, дискретных, очевидных задач. Теперь вы можете сделать что-то вроде этого:

```
$users = [...];
$countAdmins = collect($users)->filter(function ($user) {
    return $user['status'] == 'admin';
})->count();
```

Или этого:

```
$users = [...];
$greenTeamPoints = collect($users)->filter(function ($user) {
    return $user['team'] == 'green';
})->sum('points');
```

В оставшейся части главы мы продолжим использовать в примерах воображаемую коллекцию `$users`, которую начали использовать здесь. Каждый элемент массива `$users` представляет одного пользователя и, вероятно, является доступным в виде массива. Конкретный набор свойств, которым обладает каждый пользователь, будет немного варьироваться, но во всех примерах переменная `$users` будет представлять ту коллекцию, с которой мы работаем.

Некоторые методы

Мы рассмотрели небольшую часть доступных возможностей. Рекомендую ознакомиться с полным списком доступных методов в документации по коллекциям Laravel (<http://bit.ly/2FwS1VN>), но в качестве вводной информации перечислю здесь некоторые основные методы.

- ❑ **all() и toArray().** Если нужно преобразовать коллекцию в массив, то поможет метод **all()** или **toArray()**. **toArray()** преобразует в плоские массивы не только саму коллекцию, но и все вложенные объекты Eloquent. **all()** преобразует в массив *только* саму коллекцию, оставив без изменений любые содержащиеся в ней объекты Eloquent. Вот несколько примеров:

```
$users = User::all();

$users->toArray();

/* Возвращает
   [
       ['id' => '1', 'name' => 'Agouhanna'],
       ...
   ]
*/

$users->all();

/* Возвращает
   [
       Объект Eloquent { id : 1, name: 'Agouhanna' },
       ...
   ]
*/
```

- ❑ **filter() и reject().** Чтобы получить подмножество элементов исходной коллекции путем проверки каждого элемента с помощью замыкания, подходит метод **filter()** (отбирает элемент, если замыкание возвращает значение **true**) или метод **reject()** (отбирает элемент, если замыкание возвращает значение **false**):

```
$users = collect([...]);
$admins = $users->filter(function ($user) {
    return $user->isAdmin;
});

$paidUsers = $users->reject(function ($user) {
    return $user->isTrial;
});
```

- ❑ **where().** Метод позволяет легко получить подмножество элементов исходной коллекции, у которых заданному ключу присвоено заданное значение. Хотя для всего, что можно сделать с помощью метода **where()**, подходит и метод **filter()**, это сокращенный псевдоним для распространенного сценария:

```
$users = collect([...]);
$admins = $users->where('role', 'admin');
```

- ❑ **first() и last().** Если нужно извлечь из коллекции только один элемент, то воспользуйтесь методом **first()** для извлечения элемента в начале списка или методом **last()** для извлечения элемента в конце списка.

При вызове методов **first()** и **last()** без параметров они просто возвращают первый или последний элемент коллекции. Если вы передадите этим методам замыкание, то они возвратят первый или последний элемент коллекции, *который возвращает значение true при передаче его замыканию*.

Иногда это нужно, чтобы получить фактический первый или последний элемент. С другой стороны, иногда это самый простой способ получить один элемент даже в том случае, когда коллекция содержит только один элемент:

```
$users = collect([...]);
$owner = $users->first(function ($user) {
    return $user->isOwner;
});

$firstUser = $users->first();
$lastUser = $users->last();
```

Этим методам также можно передать в качестве второго параметра значение по умолчанию, используемое в том случае, когда замыкание не возвращает никакого результата.

- ❑ **each().** Чтобы произвести над каждым элементом коллекции определенные действия, которые не включают в себя изменение элементов или самой коллекции, можно воспользоваться методом **each()**:

```
$users = collect([...]);
$users->each(function ($user) {
    EmailUserAThing::dispatch($user);
});
```

- ❑ **map().** Чтобы перебрать все элементы коллекции, внести в них изменения и вернуть новую коллекцию с модифицированными элементами, воспользуйтесь методом **map()**:

```
$users = collect([...]);
$users = $users->map(function ($user) {
    return [
        'name' => $user['first'] . ' ' . $user['last'],
        'email' => $user['email'],
    ];
});
```

- ❑ **reduce().** Чтобы получить из коллекции одно значение, представляющее собой результат некоторого подсчета или строку, воспользуйтесь методом **reduce()**. Он берет некоторое исходное значение так называемого переноса и позволяет каждому элементу коллекции определенным образом модифицировать это значение. Можно задать начальное значение переноса и замыкание, принимающее в качестве параметров текущие значения переноса и каждого элемента:

```
$users = collect([...]);
```

```
$points = $users->reduce(function ($carry, $user) {
    return $carry + $user['points'];
}, 0); // Начинаем с переноса, равного 0
```

- ❑ **pluck()**. Чтобы извлечь только значения заданного ключа, содержащиеся в каждом элементе коллекции, воспользуйтесь методом **pluck()** (**lists()** в Laravel 5.1 и более ранних версиях):

```
$users = collect([...]);
```

```
$emails = $users->pluck('email')->toArray();
```

- ❑ **chunk()** и **take()**. Метод **chunk()** позволяет легко разбить коллекцию на группы заданного размера, а метод **take()** извлекает заданное количество элементов:

```
$users = collect([...]);
```

```
$rowsOfUsers = $users->chunk(3); // Разбивает на группы по три элемента
```

```
$topThree = $users->take(3); // Извлекает первые три элемента
```

- ❑ **groupBy()**. Чтобы сгруппировать все элементы коллекции по значению одного из свойств, воспользуйтесь методом **groupBy()**:

```
$users = collect([...]);
```

```
$usersByRole = $users->groupBy('role');
```

```
/* Возвращает:
```

```
[
    'member' => [...],
    'admin' => [...],
]
```

```
*/
```

Можно передать в качестве параметра замыкание. Тогда элементы будут группироваться по результату, возвращаемому этим замыканием:

```
$heroes = collect([...]);
```

```
$heroesByAbilityType = $heroes->groupBy(function ($hero) {
    if ($hero->canFly() && $hero->isInvulnerable()) {
        return 'Kryptonian';
    }

    if ($hero->bitByARadioactiveSpider()) {
        return 'Spidermanesque';
    }

    if ($hero->color === 'green' && $hero->likesSmashing()) {
        return 'Hulk-like';
    }

    return 'Generic';
});
```

- ❑ `reverse()` и `shuffle()`. Метод `reverse()` меняет порядок элементов коллекции на противоположный, а `shuffle()` располагает их в случайном порядке:

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

$numbers->reverse()->toArray(); // [3, 2, 1]
$numbers->shuffle()->toArray(); // [2, 3, 1]
```

- ❑ `sort()`, `sortBy()` и `sortByDesc()`. Если элементы представляют собой простые строки или целые числа, то их можно отсортировать с помощью метода `sort()`:

```
$sortedNumbers = collect([1, 7, 6])->sort()->toArray(); // [1, 6, 7]
```

В случае более сложных элементов можно вызвать метод `sortBy()`, передав ему строку, представляющую свойство, по которому требуется выполнять сортировку. Или метод `sortByDesc()`, передав ему замыкание, определяющее поведение сортировки:

```
$users = collect([...]);

// Сортировка массива пользователей по свойству "email"
$users->sort('email');

// Сортировка массива пользователей по свойству "email"
$users->sortBy(function ($user, $key) {
    return $user['email'];
});
```

- ❑ `count()`, `isEmpty()` и `isNotEmpty()`. Метод `count()` позволяет узнать, сколько элементов содержит коллекция, а `isEmpty()` и `isNotEmpty()` позволяют проверить, пустая коллекция или нет:

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

$numbers->count(); // 3
$numbers->isEmpty(); // false
$numbers->isNotEmpty() // true
```

- ❑ `avg()` и `sum()`. В случае, когда элементы коллекции представляют собой числа, можно вычислить среднее значение и сумму элементов с помощью методов `avg()` и `sum()`, которые не требуют каких-либо параметров:

```
collect([1, 2, 3])->sum(); // 6
collect([1, 2, 3])->avg(); // 2
```

Но если элементы коллекции представляют собой массивы, можно передать в качестве параметра ключ того свойства, которое нужно извлечь для подсчета из каждого массива.

```
$users = collect([...]);

$sumPoints = $users->sum('points');
$avgPoints = $users->avg('points');
```



Использование коллекций за пределами Laravel

Вам настолько понравились коллекции, что вы хотели бы применять их даже в тех проектах, которые не используют фреймворк Laravel? С благословения Тейлора Отвела я выделил функциональность коллекций из фреймворка Laravel в отдельный проект с названием Collect (<http://bit.ly/2f1It7n>). Сотрудники моей компании поддерживают его в актуальном состоянии по мере выхода новых релизов Laravel.

Просто выполните команду `composer require tightenco/collect` и получите в свое распоряжение класс `Illuminate\Support\Collection`, готовый к использованию в вашем коде, вместе со вспомогательной функцией `collect()`.

Резюме

Laravel предлагает вам целый набор глобальных хелперов, которые упрощают выполнение самых разных задач. Они упрощают манипулирование массивами и строками и их проверку, генерирование путей и URL-адресов, а также обеспечивают простой и единообразный доступ к некоторой важной функциональности.

Коллекции Laravel — мощный инструмент, дополняющий язык PHP возможностью использования конвейера коллекций.

18

Экосистема инструментов Laravel

По мере развития Laravel Тейлор Отвел создал целый набор инструментов, призванных упростить жизнь и рабочие процессы Laravel-разработчиков. Хотя основная часть вошла непосредственно в ядро фреймворка, есть довольно много пакетов и SaaS-сервисов, которые, не будучи включенными в ядро Laravel, являются неотъемлемой составляющей его возможностей.

Инструменты, рассмотренные в книге

Эти инструменты уже вам знакомы. Я напому, что они собой представляют, и дам ссылку на соответствующий раздел книги. Для инструментов, которые не были рассмотрены в этой книге, найдите краткое описание и ссылку на соответствующий сайт.

Valet

Valet — это сервер локальной разработки (для операционной системы Mac OS, но также есть ответвления для Windows и Linux), который позволяет легко и быстро раздавать все свои проекты через свой браузер. Устанавливается глобально на локальном компьютере разработчика с помощью утилиты Composer.

Выполнив всего несколько команд, вы получаете в свое распоряжение Nginx, MySQL, Redis и другие серверы, раздающие все имеющиеся на компьютере Laravel-приложения через домен `.test`.

Пакет Valet подробно рассмотрен в подразделе «Laravel Valet» на с. 37.

Homestead

Homestead — это конфигурационный слой поверх Vagrant, позволяющий легко обеспечить раздачу нескольких Laravel-приложений из среды Vagrant, дружественной к Laravel.

Этот инструмент был кратко представлен в подразделе «Laravel Homestead» на с. 38.

Установщик Laravel

Установщик Laravel — это пакет, который устанавливается глобально на локальном компьютере разработчика (с помощью утилиты Composer). Предназначен для упрощения и ускорения настройки нового проекта Laravel.

Работа с ним описана в подразделе «Установка Laravel с помощью установщика Laravel» на с. 39.

Mix

Mix — это система для сборки клиентской части на базе Webpack. Данный инструмент может запускать Babel, Browsersync и предпочитаемые вами препроцессоры и постпроцессоры CSS. При этом поддерживаются такие технологии, как горячая замена модуля (HMR), разделение кода, управление версиями и т. д. Mix заменил Elixir, инструмент на базе Gulp, который использовался в Laravel для той же цели.

Более подробное описание Mix см. в разделе «Laravel Mix» на с. 177.

Dusk

Dusk — это фреймворк для тестирования клиентской части, позволяющий полностью проверить ваше приложение, JavaScript-код и все остальное. Это мощный пакет, который загружается с помощью утилиты Composer и управляет реальными браузерами с помощью ChromeDriver.

Более подробное описание Dusk см. в подразделе «Тестирование с использованием Dusk» на с. 338.

Passport

Passport — мощный, легко настраиваемый сервер OAuth2 аутентификации клиентов для доступа к вашему API. Он устанавливается в каждом приложении как пакет утилиты Composer. С небольшим усилием можно сделать доступными для своих пользователей все возможности аутентификации по протоколу OAuth2.

Подробное описание Passport см. в разделе «Аутентификация API с помощью Laravel Passport» на с. 370.

Horizon

Horizon — это пакет для отслеживания очередей, который можно установить в каждое приложение с помощью утилиты Composer. Он предлагает полнофункциональный пользовательский интерфейс для отслеживания рабочего состояния, производительности, отказов и хронологических данных очереди заданий Redis.

Более подробное описание Horizon см. в разделе «Laravel Horizon» на с. 488.

Echo

Echo — это JavaScript-библиотека (введенная вместе с рядом улучшений в системе уведомлений Laravel), которая упрощает подписку на события и каналы рассылки приложения Laravel, использующие веб-сокеты.

Подробное описание Echo см. в подразделе «Laravel Echo (сторона JavaScript-кода)» на с. 496.

Инструменты, не рассмотренные в книге

Ниже перечислен ряд не рассмотренных ранее инструментов из-за ограниченного объема книги. Некоторые из них решают специализированные задачи (Cashier — прием платежей, Socialite — аутентификация с помощью социальных сетей и т. д.). Некоторые я использую практически каждый день (в частности, Forge).

Я привожу здесь лишь краткое описание каждого инструмента, начиная с наиболее востребованных. Помните, что это еще далеко не полный список!

Forge

Forge (<https://forge.laravel.com/>) — это платный SaaS-инструмент для создания и управления виртуальными серверами в таких хранилищах, как DigitalOcean, Linode, AWS и др. Он снабжает совместимые с Laravel серверы (и отдельные сайты на этих серверах) необходимыми рабочими инструментами: от очередей и обработчиков очередей до SSL-сертификатов Let's Encrypt. Он позволяет создавать простые сценарии командной оболочки для автоматического развертывания ваших сайтов после загрузки нового кода в GitHub или Bitbucket.

Forge невероятно удобен в качестве средства быстрого и легкого развертывания сайтов. Но его возможности не настолько минимальны, чтобы его нельзя было использовать для долгосрочной или крупномасштабной эксплуатации приложений. Вы можете увеличивать масштаб своих серверов, добавлять балансировщики нагрузки и управлять приватным сетевым взаимодействием между своими серверами, не прибегая к другим инструментам.

Envoyer

Envoyer (<https://envoyer.io/>) — это платный SaaS-инструмент, призванный обеспечить «развертывание PHP-кода с нулевым временем простоя». В отличие от Forge Envoyer не развертывает ваши серверы или управляет ими. Его задача сводится

к прослушиванию запускающих событий — обычно таких, как загрузка нового кода. Запускать развертывание можно вручную или веб-хуками.

Envoyer имеет три преимущества по сравнению с Forge и большинством других решений для развертывания после загрузки кода.

1. Он предлагает надежный набор инструментов для выстраивания вашего конвейера развертывания в виде простого, но мощного многоэтапного процесса.
2. Развертывает приложение с нулевым временем простоя в стиле инструмента Capistrano. Каждое новое развертывание производится в отдельную папку, причем эта папка развертывания привязывается посредством символической ссылки к реальному корневому каталогу только после успешного завершения процесса сборки. В силу этого не требуется останавливать работу сервера на то время, пока Composer производит установку или NPM делает сборку.
3. Использование такой системы папок позволяет легко и быстро выполнять откат к предыдущему релизу в случае неудачного развертывания изменений. Envoyer просто снова направляет символическую ссылку на предыдущую папку развертывания, из которой сразу же начинается раздача предыдущей сборки.

Envoyer можно настроить так, чтобы он регулярно проверял работоспособность (путем отправки пингов вашим серверам, которые сообщают об ошибке, если пинги не возвращают HTTP-ответ с кодом 200), требовал, чтобы ему регулярно отправляли пинги задания cron, и отсылал мгновенные текстовые уведомления о любых значимых событиях.

Envoyer — это более узкоспециализированный инструмент по сравнению с Forge. Немногие из знакомых мне Laravel-разработчиков используют Forge. Если кто-то из них не пожалел денег на Envoyer, то их сайт не допускает даже малейшей задержки в откате в случае неудачного внесения изменений либо принимает настолько большой (или важный) трафик, что периодические 10-секундные простои могут представлять серьезную проблему. Если ваш сайт относится к этой категории, то с Envoyer вы почувствуете себя настоящим волшебником!

Cashier

Cashier (<http://bit.ly/2Or9V0r>) — это бесплатный пакет, который предоставляет простой интерфейс для функций оплаты подписки, предлагаемых сервисами Stripe и Braintree. Cashier поддерживает практически все простейшие функции подписки пользователей, изменения их тарифных планов, предоставления им доступа к счетам, обработки обратных вызовов веб-хуков, поступающих от платежного сервиса, управления отменой отсрочки платежа и т. д.

Если нужна возможность оформления подписки с использованием сервиса Stripe или Braintree, то Cashier может существенно облегчить вам жизнь.

Socialite

Socialite (<http://bit.ly/2TVjmvd>) — это бесплатный пакет, который упрощает добавление в ваши приложения аутентификации с помощью социальных сетей (таких как GitHub или Facebook).

Nova

Nova (<https://nova.laravel.com/>) — это платный пакет для создания панелей администратора. Если мы посмотрим, что обычно представляет собой более или менее сложное Laravel-приложение, то оно включает в себя следующие компоненты: внешний сайт или клиентское представление, раздел администрирования для внесения изменений в базовые данные или список клиентов и, возможно, API.

Пакет Nova кардинально упрощает процесс создания административной части сайта на базе Vue и API Laravel. Он позволяет легко генерировать CRUD-страницы (для создания, чтения, обновления и удаления) всем вашим ресурсам вместе с более сложными пользовательскими представлениями данных, пользовательскими действиями и связями для каждого ресурса и даже пользовательскими инструментами для добавления в ту же универсальную среду администратора инструментов, не соответствующих стилю CRUD.

Spark

Spark (<https://spark.laravel.com/>) — платный пакет для создания SaaS-приложения, принимающего платежи и позволяющего легко управлять пользователями, командами и подписками. Он предлагает интеграцию с сервисом Stripe, выставление счетов, двухфакторную аутентификацию, управление аватарами пользователей, работу с командами, сброс паролей, добавление оповещений, аутентификацию с помощью токенов API и многое другое.

Spark — одновременно и набор маршрутов, и набор компонентов Vue. Поскольку на основе Spark обычно строится весь каркас нового проекта, не стоит добавлять его в уже готовое приложение.

Lumen

Lumen (<https://lumen.laravel.com/>) — это бесплатный API-ориентированный микро-фреймворк на основе компонентов Laravel. В силу ориентации на API в его состав не включены многие из вспомогательных средств Laravel, которые не ориентированы на API-вызовы (например, шаблоны Blade).

Так получился более компактный фреймворк, который лишен некоторых удобных средств, но зато обеспечивает прирост скорости.

Мой подход к Lumen сводится к следующему: если нет необходимости ускорять созданные в Laravel API или *обязательно* создавать API в стиле микросервисов, которым никогда не потребуются какие-либо вспомогательные средства фреймворка, то лучше использовать Laravel.

Если же вы разрабатываете в Laravel API в стиле микросервисов и нужно выжать максимальную скорость на уровне миллисекунд, то лучше использовать Lumen.

Envoy

Envoy (<http://bit.ly/2CDa9Ns>) — это инструмент для запуска локальных задач. С ним можно легко определить распространенные задачи, которые будут выполняться на ваших удаленных серверах, зарегистрировать их определения в системе управления версиями и запускать простым и предсказуемым образом.

Типичная задача Envoy показана в примере 18.1.

Пример 18.1. Типичная задача Envoy

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])
```

```
@task('deploy', ['on' => ['web-1', 'web-2']])
```

```
    cd mysite.com
```

```
    git pull origin {{ $branch }}
```

```
    php artisan migrate
```

```
    php artisan route:cache
```

```
@endtask
```

Чтобы запустить задачу из примера 18.1, нужно выполнить следующую команду в локальном окне терминала:

```
envoy run deploy --branch=master
```

Telescope

Telescope (<http://bit.ly/2HQPg4B>) — бесплатный инструмент отладки, который может устанавливаться в качестве пакета в приложениях Laravel, использующих версию 5.7.7 и следующие. Он генерирует информационную панель с подробными данными о текущем состоянии заданий, обработчиков очередей, HTTP-запросов, запросов к базе данных и т. д.

Другие ресурсы

Хотя многие из этих ресурсов по Laravel уже упоминались, будет нелишним перечислить их еще раз. Но это далеко не полный список.

- ❑ Новости по Laravel (<https://laravel-news.com/>).
- ❑ Обучающие видео Laracasts (<https://laracasts.com/>).

- ❑ Страница Тейлора Отвела (<https://twitter.com/taylorotwell>) и страница @LaravelPHP (<https://twitter.com/laravelphp>) в сети Twitter.
- ❑ Курсы Адама Уотана (<https://adamwathan.me/>).
- ❑ Курсы Криса Фидао (<https://fideloper.com/>).
- ❑ Подкаст о Laravel (<http://www.laravelpodcast.com/>).
- ❑ Множество посвященных Laravel чатов. На момент написания книги их основным местом дислокации, где можно было пообщаться с Тейлором Отвелом и другими участниками, был сервер Laravel Discord (<https://discord.me/laravel>). Однако существует и ряд неофициальных Slack- (<https://larachat.co/>) и IRC-каналов (например, #laravel в сети Freenode).

Есть множество чрезвычайно полезных блогов (в частности, автор книги ведет блог на сайте <https://mattstauffer.com/>, компания Tighten — блог на сайте <https://www.tighten.co>). Много авторов прекрасных страниц в сети Twitter, талантливых разработчиков пакетов и программистов, применяющих Laravel на практике, которых я с большим уважением вношу в данный список. Это богатое, разнообразное и щедрое сообщество разработчиков, большинство из которых с удовольствием делятся с другими всем, что они узнали сами. Часто возникают сложности не с поиском полезной информации, а с тем, как найти время на то, чтобы усвоить все, что предлагает сообщество.

Я не могу перечислить здесь буквально каждого человека или ресурс, которые могут быть полезны на пути в качестве Laravel-разработчика. Начав с этого списка, вы уже получите хорошее подспорье в первых шагах по использованию Laravel.

Глоссарий

ActiveRecord. Широко используемый шаблон объектно-реляционного преобразователя базы данных, который применяется в библиотеке Eloquent фреймворка Laravel. В ActiveRecord один и тот же класс модели одновременно определяет способ извлечения/сохранения записей базы данных и способ их представления. Кроме того, каждая запись БД представляется одной сущностью в приложении, а каждая сущность в приложении отображается на одну запись базы данных.

API. В строгом смысле это *прикладной программный интерфейс* (Application Programming Interface). Однако обычно под API понимается ряд конечных точек (и инструкций по их применению), которые могут использоваться для выполнения HTTP-запросов на чтение и изменение данных извне системы. Иногда также понимается набор интерфейсов или функциональных возможностей, предоставляемых потребителям определенным пакетом, библиотекой или классом.

Artisan. Инструмент для взаимодействия с приложениями Laravel из командной строки.

beanstalkd. beanstalk — это рабочая очередь, которая отличается простотой и отлично справляется с выполнением множества асинхронных задач. Потому используется в Laravel в качестве базового драйвера очередей. beanstalkd является программой-демоном Laravel.

Blade. Используемый в Laravel движок шаблонов.

BrowserKit. Набор средств тестирования DOM-взаимодействий, который присутствовал в Laravel до версии 5.4 и, начиная с версии 5.4, предлагается в виде пакета Composer.

Carbon. PHP-пакет, который упрощает и делает более выразительной работу с датами.

Cashier. Пакет Laravel, призванный упростить, унифицировать и расширить возможности по осуществлению расчетов с помощью платежного сервиса Stripe или Braintree, особенно в случае использования подписки.

CodeIgniter. Более ранний PHP-фреймворк, послуживший источником вдохновения для Laravel.

Composer. Менеджер зависимостей для языка PHP наподобие Ruby Gems и NPM.

CSRF (cross-site request forgery — «подделка межсайтовых запросов»). Злонамеренное действие, при котором внешний сайт выполняет запросы к вашему приложению, перехватывая управление браузерами пользователей (обычно с помощью JavaScript-кода), прошедших аутентификацию на вашем сайте. Мерой защиты служит добавление токена (и проверка на его наличие на POST-стороне) для каждой формы сайта.

Dusk. Пакет Laravel для тестирования клиентской части приложения, позволяющий проверить взаимодействия с JavaScript-кодом (в основном на базе Vue) и DOM-моделью путем выполнения тестов с развертыванием браузера на основе ChromeDriver.

Echo. Инструмент Laravel, упрощающий аутентификацию веб-сокетов и синхронизацию данных.

Elixir. Старый инструмент сборки Laravel, вместо которого теперь используется Mix; обертка вокруг Gulp.

Eloquent. Используемый в Laravel ORM на базе ActiveRecord. С помощью этого инструмента определяются и запрашиваются такие объекты, как модель **User**.

Envoy. Пакет Laravel, чтобы писать сценарии для выполнения распространенных задач на удаленных серверах. Envoy предоставляет синтаксис для определения задач и серверов вместе с утилитой командной строки для выполнения этих задач.

Envoyer. SaaS-компонент Laravel для развертывания с нулевым временем простоя, многосерверного развертывания и проверки работоспособности сервера и утилиты cron.

Faker. PHP-пакет, упрощающий генерирование случайных данных. Позволяет запросить данные различных категорий, таких как имена, адреса и временные метки.

Flysystem. Пакет, используемый Laravel для упрощения своего локального и облачного доступа к файлам.

Forge. Компонент Laravel, упрощающий развертывание и администрирование виртуальных серверов в таких крупных облачных хранилищах, как DigitalOcean и AWS.

FQCN (fully qualified class name — «полностью определенное имя класса»). Имя любого заданного класса, трейта или интерфейса с полностью указанным пространством имен. Например, название класса — **Controller**, то FQCN-имя может выглядеть как **Illuminate\Routing\Controller**.

Gulp. Инструмент сборки на базе JavaScript.

HMR (Hot Module Replacement — «горячая замена модуля»). Технология, позволяющая перезагружать только определенные фрагменты зависимостей клиентской части активного сайта без перезагрузки всего файла.

Homestead. Инструмент Laravel, который обертывает Vagrant и упрощает развертывание виртуальных серверов для локальной разработки приложений Laravel наподобие серверов Forge.

Horizon. Пакет Laravel, который предлагает инструменты для управления очередями с большими широкими возможностями по сравнению со стандартными инструментами Laravel, а также отражает текущие и хронологические данные о рабочем состоянии обработчиков очередей и их заданий.

Illuminate. Пространство имен верхнего уровня для всех компонентов Laravel.

IoC (inversion of control — «инверсия управления»). Концепция, подразумевающая передачу контроля над способом создания конкретного экземпляра интерфейса на более высокий уровень кода пакета с более низкого уровня кода. В отсутствие инверсии управления каждый отдельный контроллер и класс может решать сам, какой именно объект Mailer ему следует создавать. При наличии инверсии управления низкоуровневый код — все эти контроллеры и классы — лишь запрашивает объект Mailer. Какой-то высокоуровневый конфигурационный код *однократно* определяет для всего приложения, какой экземпляр должен предоставляться для удовлетворения такого запроса.

JSON (JavaScript Object Notation — «объектная нотация JavaScript»). Синтаксис для представления данных.

JWT (веб-токен JSON). Объект JSON, содержащий всю информацию для определения статуса аутентификации и прав доступа пользователя. Для подтверждения надежности этот объект JSON снабжается цифровой подписью с использованием хеш-кода аутентификации сообщения (HMAC) или алгоритма RSA. Обычно передается в заголовке.

Mailable. Архитектурный шаблон почтового сообщения, призванный отразить функциональность отправки электронной почты одному классу адресата.

Markdown. Язык форматирования, предназначенный для изменения простого текста с получением на выходе различных выходных форматов. Широко используются для форматирования текста, который с большой долей вероятности будет обрабатываться сценарием или читаться людьми в необработанном виде — такого как файлы README системы управления версиями Git.

Memcached. Размещаемое в оперативной памяти хранилище данных, призванное обеспечить простое, но быстрое сохранение данных. Memcached поддерживает только простейший способ сохранения в формате «ключ/значение».

Middleware. Ряд оберток вокруг приложения, которые фильтруют и декорируют его входные и выходные данные.

Mix. Инструмент для сборки клиентской части на базе Webpack. Пришел на смену библиотеке Elixir в версии Laravel 5.4. Может использоваться для конкатенации, минификации и препроцессинга ресурсов клиентской части, а также многого другого.

Mockery. Библиотека, входящая в состав Laravel, которая позволяет легко имитировать PHP-классы в тестах.

Nginx. Веб-сервер, сходный с Apache.

Nova. Платный пакет Laravel для создания панелей администратора для приложений Laravel.

NPM (Node Package Manager). Централизованный веб-репозиторий для пакетов Node, расположенный по адресу npmjs.org. Утилита, используемая на локальном компьютере для установки зависимостей клиентской части проекта в каталог `node_modules` в соответствии со спецификациями, изложенными в файле `package.json`.

OAuth. Наиболее популярный фреймворк аутентификации для API. OAuth позволяет применять несколько типов допуска, каждый из которых предписывает собственную схему получения, использования и обновления потребителями так называемых токенов, служащих для их идентификации после выполнения первоначального аутентификационного опознавания.

ORM (object-relational mapper — «объектно-реляционный преобразователь»). Шаблон проектирования, позволяющий использовать объекты языка программирования для представления данных и их связей в реляционной базе данных.

Passport. Пакет Laravel, позволяющий легко добавить сервер аутентификации OAuth в приложение Laravel.

PHPSpec. PHP-фреймворк для тестирования.

PHPUnit. PHP-фреймворк для тестирования. Является наиболее часто используемым и имеет связи с большей частью пользовательского тестирующего кода Laravel.

React. JavaScript-фреймворк. Создан и поддерживается компанией Facebook.

Redis. Хранилище данных, которое, подобно Memcached, проще, но в то же время быстрее и мощнее по сравнению с большинством реляционных баз данных. Redis поддерживает очень ограниченный набор структур и типов данных, зато обеспечивает более высокую скорость и масштабируемость.

REST (Representational State Transfer, передача состояния представления). На данный момент самый распространенный формат API. Обычно подразумевает, что каждое взаимодействие с API должно выполняться независимо от других и не должно иметь состояния. Обычно подразумевается использование команд HTTP как простейшего средства различения запросов.

S3 (Simple Storage Service, сервис простого хранилища). Сервис «объектного хранилища» от компании Amazon, упрощающий использование невероятных вычислительных мощностей сервиса AWS для сохранения и раздачи файлов.

SaaS (Software as a Service, программное обеспечение как сервис). Платные веб-приложения.

Scout. Пакет Laravel для полнотекстового поиска по моделям Eloquent.

Socialite. Пакет Laravel, упрощающий добавление аутентификации с помощью социальных сетей (вроде Facebook) в приложения Laravel.

Spark. Инструмент Laravel, позволяющий легко развернуть новое SaaS-приложение на базе подписки.

Symfony. PHP-фреймворк, ориентированный на создание высококачественных публичных компонентов. Компонент HTTP Foundation фреймворка Symfony служит «фундаментом» для Laravel и всех современных PHP-фреймворков.

Telescope. Пакет Laravel для добавления вспомогательного инструмента отладки в приложения Laravel.

Tinker. Используемый в Laravel цикл «чтение — вычисление — печать» (read — evaluate — print loop, REPL). Это инструмент, позволяющий выполнять из командной строки сложные операции языка PHP в полном контексте приложения.

Vagrant. Инструмент командной строки, позволяющий легко создавать виртуальные машины на локальном компьютере, используя предварительно определенные образы.

Valet. Пакет Laravel (для операционной системы Mac OS, но также есть ответвления для macOS и Windows), который позволяет легко раздавать приложения из произвольной папки разработки, не беспокоясь о Vagrant или виртуальных машинах.

Vue. JavaScript-фреймворк, рекомендуемый к использованию совместно с Laravel. Написан Эваном Ю.

Webpack. Webpack — это инструмент, который в техническом отношении представляет собой упаковщик модулей. Используется для запуска задач сборки клиентской части, особенно тех, которые связаны с преобразованием CSS-кода,

JavaScript-кода и других исходных файлов клиентской части в более готовый к эксплуатации формат.

Автоматическое внедрение. Когда контейнер внедрения зависимостей выполняет внедрение экземпляра разрешимого класса, избавляя разработчика от необходимости явно указывать ему, как следует разрешать этот класс, это называется автоматическим внедрением. В случае контейнера без функции автоматического внедрения вы не сможете внедрить даже самый простой РНР-объект без каких-либо зависимостей, пока не выполните его явную привязку к контейнеру. Если контейнер с автоматическим внедрением, явно привязывать к контейнеру нужно лишь те классы, зависимости которых слишком сложны или смутно выражены, чтобы контейнер мог разрешить их самостоятельно.

Авторизация. Исходя из предположения, что вы успешно прошли либо не смогли пройти аутентификацию, авторизация определяет, что вам *разрешено* делать на основе ваших конкретных идентификационных данных. Авторизация имеет отношение к доступу и контролю.

Аксессор. Метод, который определяется в модели Eloquent для настройки способа возвращения некоторого свойства. Аксессоры позволяют указать, чтобы при получении от модели некоторого свойства возвращалось другое (или, скорее, по-другому отформатированное) значение, а не то значение, которое хранится в базе данных для этого свойства.

Активная загрузка. Метод загрузки, избавляющий вас от N+1 проблем за счет дополнения первого запроса вторым запросом на извлечение набора связанных элементов. Обычно у вас есть первый запрос на извлечение коллекции объектов А. Однако у каждого объекта А есть множество объектов В. Потому при каждом извлечении объектов В из А нужен новый запрос. Активная загрузка подразумевает выполнение сразу двух запросов в одном: сначала извлекаются все объекты А, а затем — *все* объекты В, связанные с этими объектами А.

Аргумент (Artisan). Аргументы — это параметры, которые могут передаваться консольным командам Artisan. Аргументы просто принимают одно значение и не требуют написания символов -- перед ними или = после них.

Аутентификация. Аутентификация — это правильная идентификация себя в качестве члена/пользователя приложения. Аутентификация определяет, *кем* вы являетесь (или не являетесь), не давая никакого определения в отношении того, *что* вы можете делать.

Валидация. Обеспечение соответствия пользовательского ввода ожидаемым шаблонам.

Внедрение зависимостей. Шаблон проектирования, в котором зависимости внедряются извне (обычно через конструктор), вместо того чтобы инстанцироваться в классе.

Директива. Элемент синтаксиса языка Blade, выглядящий как `@if`, `@unless` и т. д.

Задание. Класс, предназначенный для инкапсуляции одной задачи. Задания могут помещаться в очередь и выполняться в асинхронном режиме.

Замыкание. Замыкания — это используемая в языке PHP разновидность анонимных функций. Замыкание — функция, которую можно передавать как объект, в качестве параметра другим функциям и методам, присваивать переменной и даже подвергать сериализации.

Интеграционный тест. Интеграционные тесты служат для проверки того, как отдельные модули взаимодействуют друг с другом и передают сообщения.

Коллекция. Шаблон проектирования, а также реализующий его инструмент Laravel. Будучи более продвинутой версией массивов, коллекции предлагают операции отображения, свертки, фильтрации и многие другие мощные возможности, которых нет у обычных массивов языка PHP.

Команда. Отдельная консольная задача утилиты Artisan.

Компоновщик представлений. Инструмент, который определяет, что при каждой загрузке определенного представления оно будет снабжаться некоторым набором данных.

Контейнер. Очень объемный термин, используемый в Laravel для обозначения контейнера приложения, обеспечивающего внедрение зависимостей. Контейнер доступен через метод `app()` и также дает разрешение вызова контроллеров, событий, заданий и команд. Это своего рода клей, скрепляющий воедино каждое приложение Laravel.

Контракт. Другое название интерфейса.

Контроллер. Класс, который обеспечивает направление к сервисам и данным приложения запросов пользователей и возвращение пользователям какого-то полезного ответа.

Маршрут. Определение способа или способов перехода пользователя к веб-приложению. Маршрут представляет собой шаблонное определение наподобие `/users/5`, `/users` или `/users/id`.

Массовое присваивание. Возможность одновременной передачи большого количества параметров для создания или обновления модели Eloquent с использованием массива с ключами.

Миграция. Манипулирование состоянием базы данных, сохранение данных в которую и запуск которой выполняются из кода.

Модель. Класс, служащий для представления определенной таблицы базы данных в вашей системе. В ORM на базе ActiveRecord, таких как библиотека Eloquent

Laravel, нужен для представления отдельной записи системы и взаимодействия с таблицей базы данных.

Модульный тест. Модульные тесты предназначены для проверки небольших, сравнительно изолированных модулей — обычно классов или методов.

Мультиарендность. Обслуживание одним приложением множества клиентов, у каждого из которых, в свою очередь, есть собственные клиенты. Мультиарендность часто подразумевает предоставление каждому клиенту приложения индивидуального тематического оформления и доменного имени, которые бы позволяли пользователям этого клиента отличать его сервис от сервисов других клиентов.

Мутатор. Инструмент библиотеки Eloquent для манипулирования данными, сохраненными в свойстве модели, до их сохранения в базе данных.

Мягкое удаление. Пометка строки базы данных как удаленной без ее фактического удаления. Обычно используется в сочетании с ORM, который по умолчанию скрывает все удаленные строки.

Область видимости. В Eloquent служит инструментом для определения последовательного и простого способа сужения запроса.

Опция (Artisan). Подобно аргументам, опции представляют собой параметры, которые могут передаваться командам Artisan. Они требуют написания перед ними символов -- и могут использоваться в качестве флага (--force) или для предоставления данных (--userId=5).

Очередь. Стек, в который можно добавлять задания. Обычно связывается с обработчиком очередей, последовательно извлекающим задания из очереди, обрабатывает их, а затем отбрасывает.

Первичный ключ. Обычно таблица базы данных имеет столбец, предназначенный для представления каждой отдельной строки. Такой столбец называется первичным ключом. Ему принято присваивать имя id.

Переменные среды. Переменные, которые определяются в файле .env и подлежат исключению из системы управления версиями. Это означает, что они не включаются в число синхронизируемых между средами параметров и сохраняются в неизменном виде.

Подсказка типа. Указание имени класса или интерфейса перед именем переменной в сигнатуре метода. Дает интерпретатору PHP (а также Laravel и другим разработчикам) указание, что в качестве этого параметра можно передавать только объект указанного класса или интерфейса.

Полиморфный. В терминологии баз данных, способный взаимодействовать с множеством таблиц базы данных, имеющих аналогичные характеристики. Полиморфная связь позволяет аналогично подключать сущности различных моделей.

Представление. Отдельный файл, который принимает данные от серверной системы или фреймворка и преобразует их в HTML-код.

Препроцессор. Инструмент сборки, который принимает код, написанный на специальной разновидности языка (в случае CSS одной из таких специальных разновидностей является язык LESS) и выдает код на обычном языке (CSS). Препроцессоры включают в себя дополнительные инструменты и возможности, которых нет в ядре языка.

Сервис-провайдер. В Laravel это структура, служащая для регистрации и загрузки классов и привязок к контейнеру.

Сериализация. Процесс преобразования более сложных данных (обычно модели Eloquent) в более простой формат (в Laravel это обычно массив или формат JSON).

Событие. В Laravel это инструмент для реализации шаблона публикации/подписки или шаблона наблюдателя. Каждое событие представляет какое-то произошедшее явление. Имя события отражает, что произошло (например, `UserSubscribed` — «пользователь подписался»), а в качестве «полезной нагрузки» может быть прикреплена релевантная информация. События поддерживают механизм срабатывания и прослушивания (или публикации и подписки, если вы предпочитаете использовать эти концепции).

Тест приложения. Тесты приложений, часто называемые приемочными тестами. Служат для проверки всего поведения приложения, обычно на его внешней границе, с использованием таких средств, как инструмент для обхода DOM-модели. Как раз такой инструмент предлагает для тестирования приложений фреймворк Laravel.

Текущий. Методы, которые можно выстроить в последовательную цепочку вызовов, называют текущими. Для поддержки текущего синтаксиса каждый метод должен возвращать экземпляр, готовый к встраиванию в цепочку. Это позволяет использовать вызовы наподобие `People::where('age', '>', 14)->orderBy('name')->get()`.

Точечная нотация. Перемещение по дереву наследования с использованием символа точки (.) в качестве обозначения перехода на уровень вниз. Например, в массиве вида `['owner' => ['address' => ['line1' => '123 Main St. ']]]` имеется три уровня вложенности. Используя точечную нотацию, можно представить значение `"123 Main St."` как `"owner.address.line1"`.

Уведомление. Инструмент Laravel для пересылки одного сообщения одному или множеству получателей через множество каналов уведомлений (таких как электронная почта, Slack, СМС).

Утверждение. В тестировании утверждение является базовым элементом теста. Вы проверяете *утверждение*, что одно значение должно равняться другому (быть

меньше/больше) или в каком-то массиве нужно заданное количество элементов, и т. д. Утверждения отличаются тем, что могут давать удачный или неудачный результат.

Фабрика моделей. Инструмент для определения способа генерирования приложением экземпляра модели для тестирования/заполнения данными. Обычно используется в сочетании с таким генератором поддельных данных, как `Faker`.

Фасад. Инструмент `Laravel`, призванный упростить доступ к сложным инструментам. Фасады обеспечивают статический доступ к основным сервисам `Laravel`. Поскольку за каждым фасадом стоит класс в контейнере, любой вызов вида `Cache::put()`; можно заменить двухстрочным вызовом `$cache = app('cache');`; `$cache->put()`;

Фасады реального времени. Отличаются от обычных фасадов тем, что не требуют отдельного класса. Фасады реального времени предоставляют возможность вызывать методы любого класса как статические методы, для чего нужно импортировать этот класс с добавлением префикса `Facades\` перед его пространством имен.

Флаг. Параметр в любом месте, способный находиться в двух состояниях: включен или выключен (содержащий логическое значение).

Хелпер (helper). Глобально доступная функция PHP, которая упрощает некоторую другую функциональность, — например, `array_get()` упрощает логику извлечения результатов из массивов.

Об авторе

Мэтт Стаффер — разработчик и преподаватель. Он совладелец и технический директор компании Tighten (<https://tighten.co/>), ведет блог на сайте <http://mattstauffer.com>, администрирует подкаст по Laravel и серию видео под названием «Пятиминутное шоу от компьютерного фаната» (Five-Minute Geek Show).

Об обложке

На обложке изображен орикс, или сернобык (*Oryx gazella*). Это вид больших антилоп, обитающих в пустынях Южной Африки, Ботсваны, Зимбабве и Намибии. Он представлен на национальном гербе Намибии.

Рост сернобыков в холке составляет около 1,2 м, а вес — от 100 до 300 кг. За исключением нижней части тела, их окраска коричневато-бежевая, с заметными черными полосами по бокам и на верхних частях конечностей. Впечатляющие прямые рога сернобыкам нужны для обороны — они достигают в среднем 0,8 м в длину и используются многими народами в качестве амулетов. В средневековой Англии их часто продавали под видом рогов единорога.

Хотя рога сернобыков делают их ценным объектом трофейной охоты, численность этих животных остается стабильной во всей Южной Африке. В 1969 году ориксы были завезены в южную часть американского штата Нью-Мексико. К настоящему времени их численность там составляет около 3000.

Сернобыки хорошо приспособлены к условиям таких пустынных районов, потому что могут обходиться без воды в течение значительной части года. Это удается в силу того, что им не свойственно часто дышать или потеть, поэтому в жаркие дни температура тела может подниматься на несколько градусов выше нормы. Продолжительность жизни в естественных условиях — около 18 лет.

Многие виды животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, хотя все они важны для нашего мира. По адресу <http://animals.oreilly.com> можно узнать подробнее о том, как вы можете помочь спасению этих видов животных.

Иллюстрация для обложки выполнена Карен Монтгомери на основе черно-белой гравюры, приведенной в книге *Riverside Natural History*.

Laravel

Полное руководство

Что отличает Laravel от других PHP-фреймворков? Скорость и простота. Стремительная разработка приложений, обширная экосистема и набор инструментов Laravel позволяют быстро создавать сайты и приложения, отличающиеся чистым удобочитаемым кодом.

Мэтт Стаффер, известный преподаватель и ведущий разработчик, предлагает как общий обзор фреймворка, так и конкретные примеры работы с ним. Опытным PHP-разработчикам книга поможет быстро войти в новую тему, чтобы реализовать проект на Laravel. В издании также раскрыты темы Laravel Dusk и Horizon, собрана информация о ресурсах сообщества и других пакетах, не входящих в ядро Laravel.

«Изучить фреймворк бывает непросто, но эта книга написана так, словно за вашим плечом стоит опытный товарищ, готовый подсказать и подбодрить в нужный момент».

Саманта Гейтц,
старший fullstack-разработчик в Shelterluv

В этой книге вы найдете:

- Инструменты для сбора, проверки, нормализации, фильтрации данных пользователя.
- Blade, мощный пользовательский шаблонизатор Laravel.
- Выразительную модель Eloquent ORM для работы с базами данных приложений.
- Информацию о роли объекта Illuminate Request в жизненном цикле приложения.

- PHPUnit, Mockery и Dusk для тестирования вашего PHP-кода.
- Инструменты для написания JSON и RESTful API.
- Интерфейсы для доступа к файловой системе, сессиям, куки, кэшам и поиску.
- Реализации очередей, заданий, событий и публикации событий WebSocket.



Заказ книг:
тел.: (812) 703-73-74
books@piter.com



instagram.com/piterbooks



youtube.com/ThePiterBooks



vk.com/piterbooks



facebook.com/piterbooks

WWW.PITER.COM

каталог книг и интернет-магазин

ISBN: 978-5-4461-1396-5



9 785446 113965