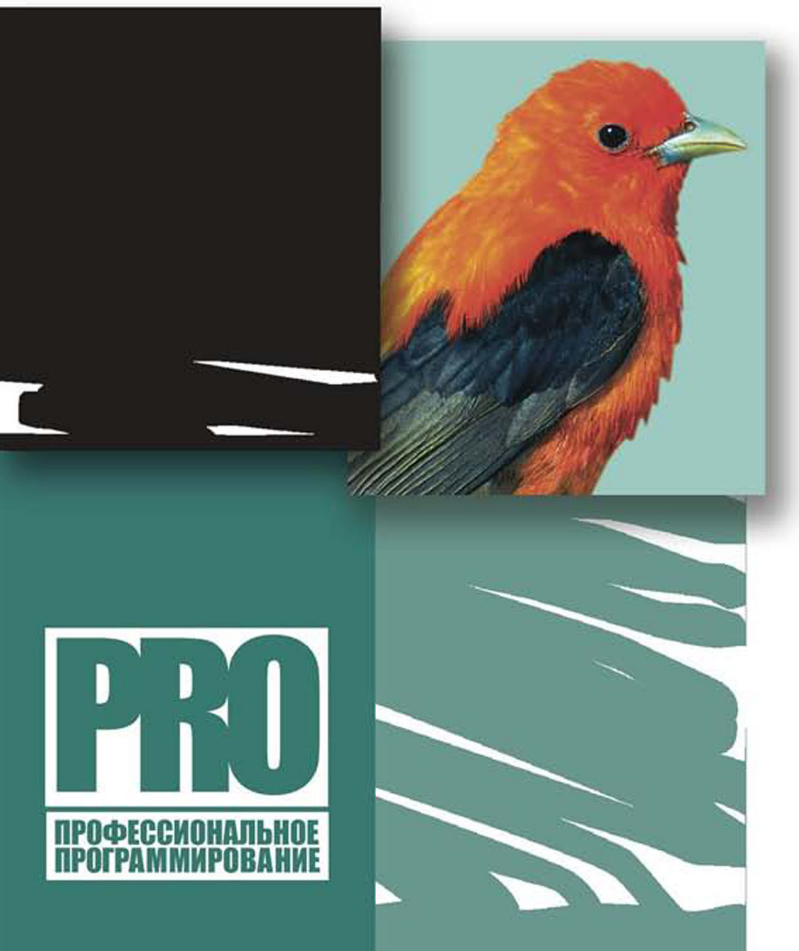
ВЛАДИМИР ДРОНОВ

LARAVEL

Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS



МИГРАЦИИ

КОМАНДЫ BLADE

ВАЛИДАЦИЯ ДАННЫХ

ВЫГРУЗКА ФАЙЛОВ

РАЗГРАНИЧЕНИЕ ДОСТУПА

ИСПОЛЬЗОВАНИЕ CAPTCHA

ПОДДЕРЖКА BBCode

ПУБЛИКАЦИЯ САЙТА

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+

Материалы на www.bhv.ru

Владимир Дронов

LARAVEL

Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS

Санкт-Петербург
«БХВ-Петербург»
2017

УДК 004.738.5+004.43
ББК 32.973.26-018.2
Д75

Дронов В. А.

Д75 Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS. — СПб.: БХВ-Петербург, 2017. — 768 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-3845-9

Книга посвящена быстрой разработке профессиональных динамических Web-сайтов с применением популярного PHP-фреймворка Laravel. Описаны технологии создания клиентской части сайта HTML 5, CSS 3 и JavaScript, а для серверной части сайта — язык PHP и сервер данных MySQL. Рассказано о применении миграций Laravel для создания в базе данных таблиц, полей, индексов и связей, о написании моделей, маршрутов, контроллеров и шаблонов. Описаны средства Laravel для ввода и правки данных, встроенные во фреймворк средства валидации с применением запросов форм и инструменты для выгрузки файлов на сайт. Рассказано о подсистеме разграничения доступа Laravel и ее настройке под конкретные нужды, а также об использовании CAPTCHA. Даны практические примеры по разработке дизайна страниц, интерактивных элементов — спойлера, лайтбокса и блокнота, создания универсального файлового хранилища, основанного на технологии AJAX, и реализации поддержки тегов BBCode для форматирования текста. Рассмотрен процесс разработки полнофункционального сайта и его публикации в Интернете. Все исходные коды доступны для загрузки с сайта издательства.

Для Web-программистов

УДК 004.738.5+004.43
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

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

ISBN 978-5-9775-3845-9

© ООО "БХВ", 2017
© Оформление. ООО "БХВ-Петербург", 2017

Оглавление

Введение в быструю разработку сайтов 1
Что придется сделать разработчику? 1
Упрощение разработки серверной части Web-сайта. Фреймворки 2
Фреймворк Laravel — номер один в Web-программировании! 3
О чем эта книга? 4
Типографские соглашения 5
Что нас ждет в будущем? 6

ЧАСТЬ I. РАЗРАБОТКА КЛИЕНТСКОЙ ЧАСТИ WEB-САЙТА 9

РАЗДЕЛ 1. СОДЕРЖИМОЕ WEB-СТРАНИЦ. ЯЗЫК HTML 5 11

Глава 1. Современный Web-дизайн. Введение в язык HTML 5 13
Клиентская часть Web-сайта 13
Статические Web-страницы и Web-сайты 14
Содержание, представление и поведение Web-страниц 14
Интернет: как все это работает? 15
  Web-серверы 15
  Интернет-адреса 16
Введение в язык HTML 5 18
  Первая Web-страница 18
  Теги и атрибуты тегов 19
  Вложенность тегов 21
Форматирование Web-страниц 22
  Секции Web-страницы 22
  Метаданные и метатеги 23

Глава 2. Структурирование и оформление текста. Литералы. Комментарии HTML 25
Структурирование текста 25
  Абзацы и заголовки 25
  Блочные элементы HTML 27
  Списки 27
  Блочные цитаты и адреса 29
  Текст фиксированного формата 30
  Блочные контейнеры 31
  Семантическая разметка текста 31
  Горизонтальные линии 33
Оформление текста 33
  Выделение фрагментов текста 33
  Встроенные элементы HTML 34
  Встроенные контейнеры 35
  Разрыв строки 35
Вставка специальных символов. Литералы 36
Комментарии 38

Глава 3. Графика и мультимедиа ... 39
Внедренные элементы Web-страниц ... 39
Графика ... 39
Поддерживаемые форматы интернет-графики ... 40
Вставка графических изображений ... 41
Мультимедиа ... 42
Поддерживаемые форматы интернет-мультимедиа ... 42
Вставка аудиоролика ... 43
Вставка видеоролика ... 44

Глава 4. Таблицы ... 46
Создание таблиц ... 46
Дополнительные инструменты для создания таблиц ... 49
Заголовок таблицы ... 49
Секции таблицы ... 50
Колонки и группы колонок ... 51
Объединение ячеек таблиц ... 53

Глава 5. Средства навигации ... 56
Текстовые гиперссылки ... 56
Создание гиперссылок ... 56
Интернет-адреса в WWW ... 58
Почтовые гиперссылки ... 59
Дополнительные возможности гиперссылок ... 59
Графические гиперссылки ... 61
Изображения-гиперссылки ... 61
Изображения-карты ... 61
Панель навигации ... 63
Якоря ... 64

Глава 6. Web-формы и элементы управления. Фреймы ... 65
Web-формы ... 65
Что такое Web-форма и зачем она нужна? ... 65
Создание Web-форм ... 66
Элементы управления ... 68
Общие вопросы создания элементов управления ... 68
Поле ввода ... 69
Поле ввода пароля ... 70
Поле ввода числа ... 70
Поле ввода интернет-адреса ... 71
Поле ввода адреса электронной почты ... 71
Флажок ... 71
Переключатель ... 72
Регулятор ... 72
Область редактирования ... 73
Список ... 73
Поле ввода файла ... 75
Скрытое поле ... 76
Кнопки ... 77

Элементы оформления ... 78
Надпись ... 78
Группа ... 78
Фреймы ... 79

РАЗДЕЛ 2. ПРЕДСТАВЛЕНИЕ WEB-СТРАНИЦ. КАСКАДНЫЕ ТАБЛИЦЫ СТИЛЕЙ CSS 3 ... 81

Глава 7. Введение в CSS ... 83
Понятие о стилях CSS ... 83
Создание стилей CSS ... 84
Таблицы стилей. Встроенные стили ... 86
Правила каскадности и приоритет стилей ... 88
Наследование атрибутов стилей ... 90
Комментарии CSS ... 91

Глава 8. Селекторы стилей. Единицы измерения CSS ... 92
Селекторы стилей ... 92
Введение в селекторы стилей ... 92
Компоненты указателей ... 93
Основные указатели ... 93
Указатели на атрибуты тега ... 95
Псевдоклассы ... 96
Псевдоэлементы ... 100
Разделители ... 100
Единицы измерения и вычисления CSS ... 101
Важные атрибуты стилей ... 103

Глава 9. Параметры текста ... 104
Параметры шрифта ... 104
Параметры вывода текста ... 108
Параметры списков ... 109
Дополнительные параметры текста ... 111
Тень у текста ... 111
Вывод текста ... 112
Загружаемые шрифты. Директивы CSS ... 113

Глава 10. Отображение и видимость элементов. Параметры курсора. Генерируемое содержание ... 116
Параметры отображения ... 116
Параметры курсора ... 118
Генерируемое содержание ... 119
Статичное генерируемое содержание ... 119
Создание нумерации ... 120

Глава 11. Параметры фона. Градиентные фоны CSS 3 ... 123
Сплошной фон ... 123
Графический фон ... 124
Создание графического фона ... 124
Параметры графического фона ... 125

Задание сразу всех параметров фона 128
Градиентный фон 128
Введение в градиенты и градиентные фоны 128
Создание линейного градиента 129
Создание радиального градиента 130
Создание повторяющегося градиента 132

Глава 12. Размеры, отступы, рамки, тени и выделение. Параметры таблиц 133
Параметры отступов 133
Параметры рамки 135
Параметры размеров 139
Параметры переполнения. Элементы с прокруткой 140
Параметры тени у блочного элемента 142
Параметры выделения 142
Параметры таблиц 143
Параметры выравнивания 144
Параметры отступов и рамок 144
Параметры размеров 146
Прочие параметры 148

Глава 13. Инструменты для создания разметки 149
Плавающие элементы 149
Позиционируемые элементы 151
Понятие позиционируемого элемента 151
Создание позиционируемых элементов 152
Гибкая верстка 155
Реализация гибкой верстки 155
Выравнивание позиций 157
Выравнивание отдельных позиций 160
Управление размерами и порядком следования позиций 162
Многоколоночная верстка 164
Базовые средства многоколоночной верстки 164
Задание дополнительных параметров колонок 165
Разметка Web-страницы и ее создание 167
Табличная разметка на основе блоков 168
Табличная разметка с фиксированными «шапкой» и «поддоном» 169
Рамочная разметка 171

Глава 14. Специальные эффекты CSS 3 174
Преобразования 174
Как задаются преобразования и их параметры? 174
Двухмерные преобразования 174
Смещение 175
Масштабирование 175
Наклон 176
Поворот 176
Трехмерные преобразования 177
Перспектива 177
Указание трехмерных преобразований 178

Точка зрения и ее местоположение 178
Скрытие обратной стороны элемента 180
Режим проецирования элементов-потомков 181
Позиционирование точки начала координат 182
Сложные преобразования 183
Анимация 183
Трансформационная анимация 183
Простейшая анимация 184
Обратная анимация 187
Сложная анимация 187
Покадровая анимация 188
Состояния анимации 189
Параметры анимации 190

Глава 15. Медиазапросы. Управление выводом на печать 194
Использование медиазапросов 194
Медиазапросы HTML 195
Введение в медиазапросы HTML 195
Указатели медиазапросов 196
Разделители медиазапросов 198
Медиазапросы CSS 199
Управление выводом на печать 199

РАЗДЕЛ 3. ПОВЕДЕНИЕ WEB-СТРАНИЦ. WEB-СЦЕНАРИИ 201

Глава 16. Язык программирования JavaScript 203
Основные понятия JavaScript 203
Типы данных JavaScript 205
Переменные 207
Именование переменных 207
Объявление переменных 207
Операторы 208
Арифметические операторы 208
Оператор объединения строк 209
Операторы присваивания 209
Операторы сравнения 210
Логические операторы 210
Оператор получения типа *typeof* 211
Преобразование типов данных 212
Приоритет операторов 213
Блоки 215
Управляющие конструкции 215
Ветвление 215
Оператор ветвления *?* 217
Переключение 217
Циклы 218
Цикл со счетчиком 218
Цикл с предусловием 219
Цикл с постусловием 220
Прерывание и перезапуск цикла 220

Функции 221
Объявление функций 221
Функции и переменные. Локальные переменные 222
Вызов функций 222
Присваивание функций. Функциональный тип данных 223
Массивы 224
Ссылки. Пустая ссылка *null* 226
Объекты и экземпляры объектов 227
Понятия объекта и экземпляра объекта 227
Создание экземпляра объекта 227
Работа с экземпляром объекта 228
Добавленные свойства и методы 229
Статические свойства и методы 229
Встроенные объекты языка JavaScript 229
Объект *Object* и использование его экземпляров 231
Оператор *instanceof* 232
Цикл по свойствам объекта 233
Обработка исключений 234
Комментарии JavaScript 236
Как Web-сценарии помещаются в код Web-страницы? 236

Глава 17. Доступ к элементам страницы и управление ими 239
Объектная модель документа 239
Доступ к странице и ее элементам 241
Доступ к странице 241
Доступ к элементам страницы 241
Прямой доступ к элементу страницы 241
Доступ по имени тега или стилевого класса. Коллекции 242
Доступ по селекторам CSS 243
Доступ к родителю, потомкам и соседним элементам. Узлы 244
Быстрый доступ к элементам страницы 246
Работа со страницей и ее элементами 247
Работа с параметрами страницы 247
Работа с параметрами элемента 247
Работа с основными параметрами 247
Работа с параметрами местоположения и размеров элемента страницы 248
Работа с атрибутами тега и их значениями 250
Работа со стилями 251
Работа с содержимым элемента 253
Добавление нового содержимого 254
Добавление и удаление элементов страницы 254

Глава 18. Обработка событий 258
Введение в события и их обработку 258
События 258
Обработчики событий и их привязка 259
Получение сведений о событии 260
События, поддерживаемые элементами страницы 261
События мыши 261
События клавиатуры 264

Событие прокрутки 265
События секции тела страницы 265
Особые случаи обработки событий 266
Туннелирование и всплытие событий 266
Обработчик события по умолчанию и его отмена 268

Глава 19. Управление интерактивными и внедренными элементами 270
Интерактивные элементы 270
Гиперссылки 270
Web-формы 271
Элементы управления 272
Внедренные элементы 279
Графические изображения 279
Аудио- и видеоролики 280

Глава 20. Работа с Web-обозревателем 285
Окна Web-обозревателя 285
Интернет-адрес текущей страницы 289
Список истории Web-обозревателя 290
Параметры экрана..291
Сведения о Web-обозревателе 292
Стандартные диалоговые окна и сообщения 293
Таймеры..294

Глава 21. Работа с локальными файлами. Регулярные выражения 296
Работа с локальными файлами 296
Получение выбранных файлов и сведений о них 296
Загрузка выбранных файлов 298
Регулярные выражения 301
Написание регулярных выражений. Литералы и группы 301
Работа с регулярными выражениями 304

Глава 22. AJAX 308
Введение в AJAX 308
Программная реализация AJAX 309
Объект *XMLHttpRequest* 309
Отправка запроса 309
Получение результата..312
Формат JSON 313
AJAX-навигация 315

ЧАСТЬ II. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ WEB-САЙТА 319

РАЗДЕЛ 4. ВВЕДЕНИЕ В СЕРВЕРНОЕ ПРОГРАММИРОВАНИЕ. PHP. MYSQL 321

Глава 23. Серверные программы. Фреймворки 323
Динамические страницы и сайты 323
Разработка серверных программ. Фреймворки 325
Введение во фреймворки. Модели, шаблоны, контроллеры 326

Глава 24. Программная платформа PHP 330
Основные принципы, типы данных, переменные и операторы 330
Управляющие конструкции 332
Функции 333
Массивы. Ассоциативные массивы 334
Регулярные выражения 335
Классы и объекты 336
    Доступ к свойствам и методам объекта 336
    Объявление классов 337
    Наследование классов 338
    Конструкторы и деструкторы 340
    Модификаторы доступа 341
    Константы класса 342
Интерфейсы 342
Трейты 343
Пространства имен 344
    Определение пространств имен 344
    Работа с пространствами имен 345
Принципы написания программного кода PHP 346

Глава 25. Базы данных. Сервер данных MySQL 348
Реляционные базы данных 348
    Введение в реляционные базы данных 348
    Поля 349
    Индексы и ключи 350
    Связи 352
Язык SQL 353
СУБД MySQL 355
    Типы данных, поддерживаемые MySQL 355
    Атрибуты полей и индексов 357
    Агрегатные функции 357
    Пользователи и их права 358

РАЗДЕЛ 5. ФРЕЙМВОРК LARAVEL 361

Глава 26. Установка и настройка Laravel 363
Программные требования Laravel 363
Создание нового проекта 363
Структура папок Laravel-проекта 365
Настройка сайта 367
    Настройка соединения с базой данных 368
    Настройки отправки электронной почты 368
    Настройки режима работы сайта 369
    Прочие настройки 370
Обработка ошибок 370

Глава 27. Миграции 372
Преимущества миграций 372
Создание миграций 373
    Прототипирование миграций 373

Код новой миграции. Фасады Laravel 374
Создание структур данных 375
Создание таблиц 375
Создание полей 375
Создание индексов 378
Создание связей 379
Правка и переименование структур данных 381
Добавление полей 381
Правка и переименование полей 382
Переименование таблиц 382
Удаление структур данных 383
Удаление связей 383
Удаление индексов 383
Удаление полей 384
Удаление таблиц 384
Выполнение и откат миграций 385
Дополнительные возможности миграций 385

Глава 28. Модели 387
Модели Laravel: требования и соглашения 387
Создание простых моделей 388
Прототипирование моделей. Базовый класс модели 388
Задание параметров модели 389
Создание связей 390
Связь «один-ко-многим» 390
Связь «один-к-одному» 393
Связь «многие-ко-многим» 393
Сквозная связь 395
Расширение функциональности модели 397
Создание вычисляемых полей 397
Указание другого поля для поиска при внедрении модели 398
Создание обработчиков событий 398
Произвольные свойства и методы модели 401

Глава 29. Маршрутизация 403
Введение в маршрутизацию 403
Где хранятся настройки маршрутизации? 404
Указание маршрутов 405
Простые маршруты 405
Параметризованные маршруты 406
Правила для значений параметров в параметризованных маршрутах 407
Именованные маршруты 409
Указание посредников для маршрутов 409
Массовое создание маршрутов 410
Базовые средства для массового создания маршрутов 410
Дополнительные параметры массово создаваемых маршрутов 412
Внедрение модели в контроллер 413
Неявное внедрение модели 413
Явное внедрение модели 414

Группы маршрутов 416
Физические интернет-адреса 417

Глава 30. Контроллеры и действия 419
Контроллеры Laravel: требования и соглашения 419
Создание контроллеров 420
Получение данных от посетителя 421
Работа с базой данных 423
Простая выборка записей 424
Поиск записей по их номерам 424
Выборка всех записей 425
Выборка первой записи 425
Получение значений полей записи 425
Получение связанных записей 425
Создание запросов к базе данных 427
Фильтрация записей 427
Фильтрация по наличию или отсутствию связанных записей 431
Сортировка записей 432
Указание выбираемых полей 433
Выборка уникальных записей 434
Связывание таблиц 434
Использование агрегатных функций. Группировка записей 435
Получение количества связанных записей 435
Использование агрегатных функций применительно ко всем записям 436
Использование агрегатных функций применительно к сгруппированным записям 437
Фильтрация и сортировка групп записей 437
Ограничение количества выбираемых записей 438
Специальные случаи выборки записей 439
Использование пагинатора 439
Упрощенный пагинатор 440
Полнофункциональный пагинатор 442
Получение сведений о запросе 442
Получение путей к папкам фреймворка 444
Вывод данных 444
Вывод посредством шаблона 445
Вывод в формате JSON 446
Вывод пагинатора в формате JSON 447
Отправка файлов 447
Перенаправление 448
Перенаправление с выводом всплывающих сообщений 449
Указание посредников в контроллерах 450
Особые разновидности контроллеров 451
Контроллеры-функции 451
Контроллеры-действия 452

Глава 31. Шаблоны 453
Шаблоны Laravel: требования и соглашения 453
Создание шаблонов 453

Команды языка Blade 456
Команды вывода данных 456
Ветвления 456
Циклы 457
Прерывание и перезапуск цикла 457
Служебная переменная *loop* 458
Комментарии Blade 459
Вставка PHP-кода 459
Особые случаи вывода данных 459
Генерирование интернет-адресов 459
Создание Web-форм и элементов управления 461
Указание метода отправки данных 461
Защита от сетевых атак 461
Вывод введенных ранее данных 462
Вывод сообщений об ошибках 462
Вывод всплывающих сообщений 463
Вывод пагинатора 463
Вложенные шаблоны 465
Наследование шаблонов 467
Создание шаблонов-родителей 467
Создание шаблонов-потомков 468
Стеки 469
Разделяемые данные и составители 470
Разделяемые данные 470
Составители 471
Получение доступа к контроллеру 473

Глава 32. Ввод и правка данных 474
Создание, правка и удаление отдельных записей 474
Создание и правка записей 474
Создание Web-формы для ввода и правки записи 474
Создание записи 476
Правка записи 479
Удаление записей 481
Обработка связей между таблицами 482
Связь «один-ко-многим» 482
Связь "многие-ко-многим" 483
Дополнительные инструменты для создания и правки отдельных записей 485
Поиск или создание записей 485
Исправление или создание записей 486
Работа со связанными записями 487
Проверка введенных в форму данных на корректность. Валидаторы 489
Простейшие валидаторы 489
Полностью автоматическая валидация 489
Полуавтоматическая валидация 490
Условные правила валидации 492
Правила валидации 492
Запросы форм 495

Массовые создание, правка и удаление записей 497
Работа с выгруженными файлами 499
Файловое хранилище и диски Laravel 499
Особенности создания Web-формы для выгрузки файлов 501
Получение и сохранение выгруженных файлов 501
Получение выгруженного файла и сведений о нем 501
Сохранение выгруженного файла 502
Работа с выгруженными файлами 503

Глава 33. Разграничение доступа. Использование CAPTCHA 505
Встроенная подсистема разграничения доступа Laravel 505
Ограничение доступа к страницам 507
Простые случаи ограничения доступа 507
Ограничение доступа на основе более сложных условий 508
Шлюзы 509
Политики 510
Ограничение доступа на основе запросов форм 514
Ограничение доступа в шаблонах 514
Вывод сведений о текущем пользователе 515
Настройка встроенной подсистемы разграничения доступа 516
Базовые настройки 516
Модификация списка пользователей 521
Настройка писем, отправляемых действием восстановления пароля 522
Создание оповещения 522
Подготовка шаблона для оповещения 525
Отправка оповещений 525
Использование CAPTCHA. Библиотека Captcha for Laravel 526
Установка 526
Настройка 527
Использование 529

Глава 34. Кэширование 530
Настройки кэширования 530
Кэширование в папке 530
Кэширование в таблице базы данных 531
Кэширование в оперативной памяти 532
Занесение данных в кэш 532
Изменение данных, хранящихся в кэше 533
Получение данных из кэша 533
Простое получение данных 533
Получение данных из кэша с одновременной их записью 534
Получение данных из кэша с последующим их удалением 535
Удаление данных из кэша 535
Работа с другим хранилищем 536

ЧАСТЬ III. ПРАКТИКА РАЗРАБОТКИ: СОЗДАНИЕ WEB-САЙТА ЭЛЕКТРОННЫХ ПУБЛИКАЦИЙ 537

РАЗДЕЛ 6. РАЗРАБОТКА WEB-САЙТА — СВОДИМ ВСЕ ВОЕДИНО 539

Глава 35. Планирование и предварительные действия 541

Планирование Web-сайта 541

Основные этапы планирования Web-сайта 541

Дизайн Web-сайта 542

Логическая и физическая структуры Web-сайта 543

«СЭП» — Web-сайт электронных публикаций 544

Дизайн сайта 544

Создание и настройка проекта 545

Глава 36. Создание дизайна Web-страниц 548

Особенности создания представления для Web-страниц 548

Web-страницы для традиционных компьютеров 549

Разметка 549

Начальное представление 550

«Шапка» 551

Панель навигации 553

Блок основного содержания 555

Параметры самого блока основного содержания 555

Параметры текста 555

Параметры нумерации заголовков 556

Параметры внедренных элементов 557

Параметры Web-форм и элементов управления 559

«Поддон» 562

Web-страницы для мобильных устройств 563

Разметка 563

«Шапка» 563

Блок основного содержания 564

«Поддон» 565

Печатная редакция Web-сайта 566

Глава 37. Интерактивные элементы 568

Спойлер 568

Формирование спойлера 568

Представление спойлера 569

Программирование спойлера 571

Лайтбокс 573

Формирование лайтбокса 573

Представление лайтбокса 574

Программирование лайтбокса 577

Блокнот 579

Формирование блокнота 579

Представление блокнота 580

Программирование блокнота 582

Глава 38. Статические Web-страницы ... 585
Маршруты ... 585
Контроллеры ... 585
Базовый класс контроллера. Определение обращения с мобильного устройства ... 586
Контроллер *MainController* ... 588
Шаблоны ... 588
Родительские шаблоны ... 588
Шаблоны страниц ... 591
Тестирование «мобильной» редакции Web-сайта ... 592

Глава 39. Разграничение доступа и список пользователей ... 594
Маршруты ... 594
Миграция и модель ... 595
Служебные страницы ... 596
Регистрация нового пользователя ... 596
Вход на Web-сайт ... 598
Процедура сброса пароля ... 599
Отправка письма со сведениями о сбросе пароля ... 599
Собственно сброс пароля ... 599
Электронное письмо со сведениями для сброса пароля ... 600
Инструменты для работы со списком пользователей ... 603
Список пользователей ... 603
Правка пользователя ... 606
Удаление пользователя ... 608
Разграничение доступа ... 610
Панель навигации ... 611

Глава 40. Категории и подкатегории ... 613
Маршруты ... 613
Миграции и модели ... 614
Миграция и модель списка категорий ... 614
Миграция и модель списка подкатегорий ... 616
Инструменты для работы со списками категорий и подкатегорий ... 617
Список категорий ... 617
Вывод списка категорий ... 617
Автоматическое генерирование слагов ... 619
Правка списка категорий ... 620
Список подкатегорий ... 623
Вывод списка подкатегорий ... 623
Создание и правка подкатегорий ... 624
Разграничение доступа ... 624
Панель навигации ... 625
Вывод списков категорий и подкатегорий ... 626

Глава 41. Статьи. Поддержка BBCode ... 631
Маршруты ... 631
Миграции и модели ... 632
Миграция и модель списка статей ... 632
Модели списков подкатегорий и пользователей ... 634

Разграничение доступа 634
Вывод списка статей 635
Вывод списка статей, относящихся к выбранной подкатегории 635
Вывод списка последних пяти статей, относящихся к выбранной категории 641
Поиск статей 642
Вывод статьи 645
Форматирование текста статей. BBCode 645
Набор тегов BBCode, поддерживаемых нашим сайтом 645
Собственно форматирование текстов статей 646
Собственно вывод статьи 649
Инструменты для работы над статьями 652

Глава 42. Комментарии 657
Маршруты 657
Миграция и модель 657
Комментарии, относящиеся к выбранной статье 659
Страницы для работы с комментариями 663

Глава 43. Хранилище файлов 668
Маршруты 668
Миграция и модель 668
Миграция 669
Модель 670
Контроллер 671
Визуальная часть 674
Шаблон 675
Представление 676
Web-сценарий 678

РАЗДЕЛ 7. НАНЕСЕНИЕ ПОСЛЕДНИХ ШТРИХОВ И ПУБЛИКАЦИЯ WEB-САЙТА 685

Глава 44. Программируемая графика HTML 5 687
Канва 687
Контекст рисования 687
Рисование прямоугольников 688
Задание цвета, уровня прозрачности и толщины линий 688
Рисование сложных фигур 690
Как рисуются сложные контуры? 690
Перо. Перемещение пера 690
Прямые линии 691
Дуги 691
Кривые Безье 692
Прямоугольники 693
Задание стиля линий 693
Вывод текста 695
Использование сложных цветов 696
Линейный градиент 696
Радиальный градиент 698
Графический цвет 699

Вывод внешних изображений 700
Создание тени у рисуемой графики 701
Преобразования системы координат 701
Сохранение и загрузка состояния 702
Перемещение начала координат канвы 702
Поворот системы координат 703
Изменение масштаба системы координат 704
Управление наложением графики 705
Создание маски 706

Глава 45. Хранение данных на стороне клиента 707
Хранилище HTML 5 707
Временное хранение текста статей на стороне клиента 708

Глава 46. Публикация Web-сайта 711
Подготовка Web-сайта к публикации 711
Указание окончательных настроек 711
Удаление ненужных данных 712
Создание страниц с сообщениями об ошибках 714
Публикация сайта 715

Заключение 717

ПРИЛОЖЕНИЯ 719

Приложение 1. Установка и настройка пакета OpenServer 721
Установка 721
Запуск, перезапуск и остановка 722
Настройка 723
Запуск консоли OpenServer 724

Приложение 2. Работа с базами данных MySQL в программе phpMyAdmin 726
Запуск и вход 726
Работа с пользователями 727
Создание пользователя и базы данных 727
Правка и удаление пользователей 729
Работа с записями таблиц 730
Перенос содержимого из одной базы данных в другую 732
Экспорт данных 732
Импорт данных 733
Выход 733

Приложение 3. Перекодирование видеофайлов в формат MP4 734

Приложение 4. Файловый архив 738

Предметный указатель 739

Введение в быструю разработку сайтов

Рассмотрим типичную ситуацию. К разработчику сайтов приходит заказчик.

Заказчик: Мне нужен сайт. Только быстро!

Разработчик: Хорошо-хорошо! Какого рода сайт?

Заказчик: Система электронных публикаций. Чтобы посетители могли регистрироваться, писать статьи, а читатели оставлять на них комментарии. Только быстро!

Разработчик: Ясно. Сделаем следующим образом: все статьи будут храниться в базе данных, особая программа будет извлекать их из базы, преобразовывать к виду, понимаемому Web-обозревателями (то есть превращать в обычные Web-страницы), и выводить на экран.

Заказчик: Вам виднее — вы ведь разработчик… Делайте, как считаете нужным. Только быстро!

Разработчик: Нужно сделать еще разбиение статей на категории и подкатегории…

Заказчик: Да-да! Только быстро!

Разработчик: …и систему разграничения доступа, чтобы статьи могли оставлять только зарегистрированные пользователи…

Заказчик: Да-да! Только быстро!

Разработчик: …и не забыть про защиту от спамеров, например, CAPTCHA… Хорошо, сделаем!

Заказчик: Заплачу любые деньги!

Разработчик: Только быстро! Тьфу!.. прошу прощения…

Заказчик уходит. Разработчик погружается в тягостные раздумья…

Что придется сделать разработчику?

Итак, что разработчик должен сделать, чтобы угодить заказчику?

- Нарисовать макеты всех страниц, что в совокупности составят заказанный сайт.

  Такой макет представляет собой обычное графическое изображение, представляющее готовую страницу. Обычно его делают в графических пакетах, позво-

ляющих располагать отдельные фрагменты на собственных слоях (например, в Adobe Photoshop), чтобы любой из этих фрагментов можно было скопировать, сохранить в отдельном изображении и использовать в верстке.

- Сверстать страницы на основе макетов.
- Написать Web-сценарии — программы, которые встраиваются непосредственно в страницы и выполняют какие-либо действия над их содержанием в ответ на манипуляции посетителя. Так, сценарии могут подсвечивать какие-либо элементы при наведении на них курсора мыши, скрывать одни элементы и показывать другие или подгружать дополнительные данные, чтобы вывести их на экран.
- Разработать структуру базы данных, в которой будут храниться все внутренние данные сайта: списки категорий, подкатегорий, статьи, комментарии, список зарегистрированных пользователей и проч. Создать эту базу.
- Превратить сверстанные ранее страницы в своего рода заготовки, или шаблоны. Они понадобятся на следующем шаге.
- Написать серверные программы, которые будут работать совместно с Web-сервером, извлекать данные из базы и преобразовывать их в Web-страницы на основе созданных ранее шаблонов.
- Написать серверные программы, которые будут реализовывать разграничение доступа, защиту от злоумышленников и выполнять всевозможные служебные операции.
- Проверить все на предмет устойчивой работы и отсутствия ошибок.
- Опубликовать сайт в Интернете.

Сами Web-страницы (или, как говорят Web-верстальщики, их содержание) вместе с их оформлением (представлением) и сценариями (поведением) составляют клиентскую часть сайта — ее-то и будет наблюдать на экране своего компьютера посетитель. А база данных и все серверные программы — суть серверная часть сайта, «невидимая» посетителям, но не менее (если не более) важная.

Разработка клиентской части сайта — процесс весьма трудоемкий. Только на нее уйдет несколько дней.

А разработка серверной части еще более трудоемка…

Упрощение разработки серверной части Web-сайта. Фреймворки

Однако мы можем существенно упростить и, соответственно, ускорить процесс программирования серверной части. Как?

Все сайты, включающие, помимо страниц, еще и серверные программы (динамические сайты), содержат ряд программных модулей, которые выполняют одни и те же типовые задачи: работу с базой данных, вывод страниц на основе шаблонов, разграничение доступа, защиту от сетевых атак и т. п. А поскольку такие задачи оди-

наковы во всех сайтах, можно не писать эти модули каждый раз заново, а найти программный продукт, который их реализует, и сделать серверную часть сайта на его основе.

И такие программные продукты существуют! Это *фреймворки*. Уже в готовом виде, образно говоря, прямо из коробки, они готовы предоставить разработчику все, что необходимо, для:

- работы с базами данных;
- генерирования страниц на основе шаблонов;
- реализации разграничения доступа;
- защиты от сетевых атак;
- поддержки AJAX;
- и многого-многого другого.

Разработчику останется лишь установить фреймворк, сконфигурировать его (хотя бы указать сведения, необходимые для подключения к базе данных), написать и встроить в него программные модули, которые реализуют задачи, специфичные для разрабатываемого сайта: обработку конкретных данных и генерирование на их основе конкретных страниц.

Понятно, что, поскольку львиную долю работы сделали за Web-программиста разработчики фреймворка, сайт можно сделать очень быстро. Уж точно значительно быстрее, чем в очередной раз, что называется, изобретая велосипед…

Фреймворк Laravel — номер один в Web-программировании!

Фреймворков в настоящее время очень много — десятки, а может быть, и сотни. Самым же популярным продуктом такого рода является Laravel. Почему? О, у него много преимуществ!

- Laravel написан на PHP — самом популярном языке программирования динамических сайтов.
- Laravel не имеет каких-либо специфических системных требований и работает на мощностях практически всех присутствующих на рынке хостинг-провайдеров.
- Он предоставляет практически всю базовую функциональность сайтов, включая мощную и полностью готовую к работе систему разграничения доступа, средства для отправки электронной почты и исключительно удобный механизм миграций.
- Все входящие в него инструменты уже сконфигурированы наилучшим для большинства случаев образом.
- Почти все необходимое можно реализовать небольшим количеством программного кода. То есть налицо дополнительное сокращение трудоемкости.

- Существует много дополнительных библиотек, написанных сторонними разработчиками и расширяющих функциональность Laravel.
- И — сладкая вишенка на пышном торте! Laravel включает средства прототипирования — создания заготовок для программных модулей разных типов, в которые разработчику сайта останется лишь добавить свой код.

Как можно пройти мимо такой находки? Да ее любой разработчик сайтов должен тут же взять на заметку!

Возьмем на заметку и мы, пока еще новички в мире Web-программирования.

О чем эта книга?

Да-да, эта книга посвящена Laravel! И всем прочим Web-технологиям, без которых в трудном, но увлекательном, деле Web-программирования не обойтись. В их числе:

- HTML — язык для разработки самих Web-страниц, их содержания. Мы будем изучать самую актуальную его версию — HTML 5.
- CSS — язык для разработки оформления, или представления, страниц. Конкретно, CSS 3 — самую современную версию этого языка.
- JavaScript — язык программирования Web-сценариев, то есть поведения страниц.
- PHP — язык для написания серверных программ. На этом языке написан сам Laravel, следовательно, и встраиваемые в него дополнительные программные модули, реализующие специфическую для разрабатываемого сайта функциональность, пишутся также на нем.
- MySQL — самая популярная на данный момент программа, обрабатывающая базы данных.

Причем изучать эти технологии мы станем в том же порядке, в котором они здесь перечислены. А потом приступим и к самому Laravel.

А в завершение, в качестве практики, мы создадим полнофункциональный сайт — систему электронных публикаций «СЭП», предоставляющую площадку для размещения в Интернете статей на различные темы. Этот сайт можно свободно использовать как основу для разработки других, более сложных решений.

> ***Материалы полнофункционального сайта***
>
> Электронный архив с материалами сайта «СЭП: Система электронных публикаций» можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977538459.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*).

При работе над книгой автор использовал следующие версии ПО:

- Microsoft Windows 10, русскую 32-разрядную редакцию со всеми установленными обновлениями;

- OpenServer 5.2.2 — пакет хостинга, включающий в свой состав все программы, которые необходимы для запуска и отладки разрабатываемого сайта;
- Фреймворк Laravel 5.3.29.

Созданные Web-страницы проверялись в следующих Web-обозревателях:

- Microsoft Internet Explorer 11;
- Microsoft Edge 38.14393.0.0;
- Mozilla Firefox 47-50 (когда писалась книга, программа несколько раз обновлялась автоматически).

Для написания программного кода применялся текстовый редактор Notepad+, который можно найти по интернет-адресу **https://notepad-plus-plus.org/**.

Типографские соглашения

В книге будут постоянно приводиться форматы написания различных конструкций, применяемых в языках HTML, CSS, JavaScript и PHP. В них использованы особые типографские соглашения, которые мы сейчас изучим.

> ***Внимание!***
>
> Все эти типографские соглашения применяются автором только в форматах написания языковых конструкций. В реальном программном коде они не имеют смысла.

- В угловые скобки (<>) заключаются и дополнительно выделяются курсивом наименования различных значений. В реальный код, разумеется, должны быть подставлены реальные значения. Например:

  ```
  <страница или элемент>.getElementsByClassName(<имя стилевого класса>)
  ```

  Здесь вместо подстроки *<страница или элемент>* должны быть подставлены реальная страница или реальный элемент страницы, а вместо подстроки *<имя стилевого класса>* — реальное имя стилевого класса.

- В квадратные скобки (`[]`) заключаются необязательные фрагменты кода. Например:

  ```
  background-position: <горизонтальная позиция> [<вертикальная позиция>]
  ```

  Здесь *вертикальная позиция* может присутствовать, а может и отсутствовать.

- Символом вертикальной черты (`|`) разделяются параметры или значения, из которых в коде должно присутствовать лишь одно. Например:

  ```
  font-style: normal|italic|oblique
  ```

  Здесь допускается указание лишь одного из перечисленных значений — либо `normal`, либо `italic`, либо `oblique`.

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

  ```
  var lstSubcategory = document.querySelector➥
  ("select[name=subcategory]");
  ```

  Приведенный код разбит на две строки, но должен быть набран в одну. Символ ➥ при этом нужно удалить.

- Многоточием (. . .) помечены пропущенные по тем или иным причинам фрагменты кода. Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные выражения, а оставшиеся неизмененными пропущены.

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

> ***Еще раз внимание!***
>
> Все приведенные здесь типографские соглашения имеют смысл только в форматах описания конструкций языков HTML, CSS, JavaScript и PHP. В коде примеров используется лишь знак ➥ и многоточие.

Что нас ждет в будущем?

Когда работа над этой книгой уже подходила к концу, была выпущена новая версия Laravel — 5.4. Она предлагает ряд полезных нововведений: средства для написания шаблонов оповещений, подобные командам Blade, применяемым в коде шаблонов; возможность разработки компонентов, аналогичную таковой у фреймворка Yii; расширенные средства обработки коллекций записей; усовершенствования в механизмах событий и фасадов; новые посредники и проч.

Как видим, Laravel пришел в мир Web-программирования всерьез и надолго. И в более отдаленном будущем мы увидим новые версии этого замечательного фреймворка, предлагающего новые и, безусловно, замечательные возможности.

Но что делать, если Laravel все же канет в Лету? Что, если его сменит на рынке другой фреймворк, молодой да резвый? Как в свое время сам Laravel сбросил с почетного первого места в рейтинге популярности аналогичный и, кстати, замечательный продукт Yii. (Автор когда-то написал о нем книгу…)

Все (по крайней мере, большинство) Web-фреймворков построены на единых принципах. Они включают модели, осуществляющие работу с таблицами баз данных, шаблоны, на основе которых генерируются страницы, контроллеры, выполняющие задачи по выборке данных и генерированию страниц на основе этих данных и того или иного шаблона, и маршрутизатор, анализирующий полученный в запросе интернет-адрес и на основе результатов анализа определяющий, какое

действие какого контроллера следует запустить. Разница только в реализации всего этого, и она зачастую не так уж и велика.

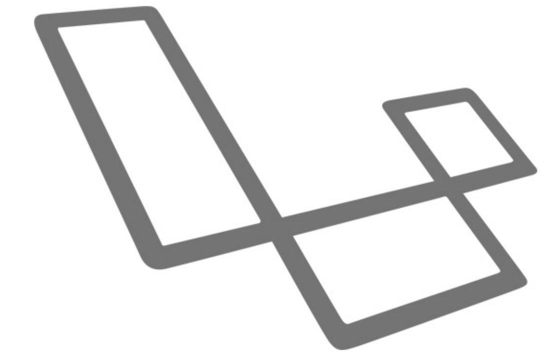
Так что мы, в случае чего, можем без проблем перейти на другой фреймворк и, после недолгого изучения документации, опять же, без особых проблем писать сайты на нем.

Но сначала нужно прочитать эту книгу. В ней есть все, что понадобится разработчику, чтобы сделать современный динамический сайт.

Быстро. Как и просил заказчик.

ЧАСТЬ I

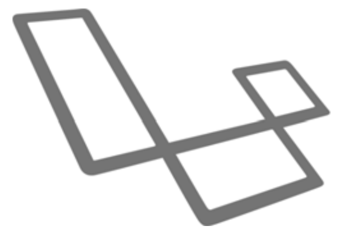
РАЗРАБОТКА КЛИЕНТСКОЙ ЧАСТИ WEB-САЙТА



I. РАЗДЕЛ 1

Содержимое Web-страниц. Язык HTML 5

Глава 1.	Современный Web-дизайн. Введение в язык HTML 5
Глава 2.	Структурирование и оформление текста. Литералы. Комментарии HTML
Глава 3.	Графика и мультимедиа
Глава 4.	Таблицы
Глава 5.	Средства навигации
Глава 6.	Web-формы и элементы управления. Фреймы



ГЛАВА 1

Современный Web-дизайн. Введение в язык HTML 5

Разработка любого приличного Web-сайта начинается с создания его клиентской части. Что это такое?

Клиентская часть Web-сайта

Клиентской на жаргоне программистов называется та часть приложения, которую пользователь непосредственно наблюдает на экране своего компьютера. Это различные окна, выводящие какие-либо данные, что были получены из базы и обработаны серверной частью.

Применительно к нашему случаю — к сайту — клиентской частью являются всевозможные Web-страницы, которые видит на экране посетитель и которые, опять же, отображают данные, полученные и обработанные серверной частью сайта (речь о которой пойдет в *части II* этой книги).

Говоря поэтично, клиентская часть — лицо сайта, в то время как часть серверная — его душа. И, как всегда, лицо является зеркалом души.

Именно поэтому первый этап создания сайта — написание его клиентской части, всех входящих в его состав Web-страниц.

Но, постойте, спросит иной читатель, зачем же писать все эти страницы, если они все равно будут генерироваться программами, входящими в состав серверной части? Зачем выполнять двойную работу?

Во-первых, на начальном этапе разрабатываются образцы, «рыбы» страниц, содержащие какие-либо произвольные данные (все равно реальная информация потом будет формироваться серверной частью). Во-вторых, программировать серверные программы удобнее, имея перед глазами готовые образцы страниц, — с этим, думается, не поспорит никто.

И, в-третьих и в-главных, современные PHP-фреймворки (мы ведь еще не передумали использовать Laravel?) выполняют формирование страниц на основе так называемых *шаблонов* (речь о них пойдет в *главе 30*). Они представляют собой обычные Web-страницы, в нужных местах кода которых — там, где должны быть под-

ставлены выводимые серверной частью данные, — имеются особые пометки. Понятно, что эти шаблоны удобнее создавать на основе уже готовых страниц.

Итак, на первом этапе нам требуется создать набор страниц, которые впоследствии станут для нас образцами при программировании серверной части сайта. То есть создать статический сайт.

Статические Web-страницы и Web-сайты

Статические страницы не генерируются серверными программами. Они верстаются Web-верстальщиком на основе подготовленного Web-дизайнером макета (представляющего собой обыкновенное графическое изображение), сохраняются в файлах, помещаются на диск серверного компьютера и все время существования сайта остаются неизменными (разумеется, до того момента, когда верстальщик не решит эти страницы переделать).

Набор взаимосвязанных статических страниц составляет *статический сайт*. Таких сайтов в Интернете довольно много — они представляют информацию, которая меняется крайне редко или не меняется вообще. Статические сайты просты и недороги в разработке и могут быть опубликованы у любого *хостинг-провайдера* (организации, предоставляющей услуги по публикации сайтов в Сети), вследствие чего популярны до сих пор.

В виде статических сайтов создается также клиентская часть полноценных динамических сайтов. Чем мы сейчас и займемся.

Содержание, представление и поведение Web-страниц

Любая страница включает в себя содержание, представление и поведение.

- *Содержание* — это, собственно, сама страница, та информация, ради которой она создана. Эта информация структурирована, то есть разбита на отдельные абзацы, заголовки, представлена в виде списков, таблиц, сгруппирована в более крупные разделы: статьи, иллюстрации, «шапки», «поддоны» и проч.
- *Представление* — это оформление страницы и отдельных ее элементов. Представление описывает, в частности, каким шрифтом будет выводиться текст абзацев и заголовков, какой отступ слева будут иметь пункты списков, какую рамку получат таблицы и даже какой эффект будет иметь наведение курсора мыши на гиперссылки.
- *Поведение* — это интерактивность страницы, ее способность реагировать на действия посетителя. Это всевозможные интерактивные эффекты, наподобие замены одной картинки на другую при наведении курсора мыши, и интерактивные элементы страницы: слайдеры, лайтбоксы, «блокноты», таблицы с возможностью прокрутки и выбора элемента и проч.

Содержание, представление и поведение, согласно современным тенденциям Web-верстки, делаются независимыми друг от друга, их даже хранят в отдельных файлах. Это позволяет упростить сопровождение сайтов (для исправления какой-либо страницы в большинстве случаев достаточно изменить лишь содержание, не трогая представление и поведение, или, напротив, изменить представление, не трогая содержание и поведение) и использовать одно и то же представление и поведение (или отдельные их части) применительно к разному содержанию.

Более того, содержание, представление и поведение создаются с применением совершенно разных технологий (о которых мы, разумеется, поговорим, но позже).

Интернет: как все это работает?

Прежде чем начать работу над клиентской частью сайта, следует хотя бы в общих чертах выяснить, как и по каким принципам работает Интернет. Мы не станем вдаваться в детали — в конце концов, мы собираемся разрабатывать сайты, а не администрировать серверы или настраивать сетевое оборудование.

Ранее упоминалось о статических сайтах, которые до сих пор часто встречаются в Интернете. Каждый такой сайт представляет собой набор файлов, хранящих входящие в его состав страницы и вспомогательные данные, и… и что дальше?

Web-серверы

Понятно, что файлы сайта хранятся на дисках компьютера, подключенного к Интернету. Понятно, что это либо компьютер, принадлежащий автору или авторам сайта, либо компьютер хостинг-провайдера. Но каким образом мы можем извлекать эти расположенные неизвестно где файлы и наблюдать их содержимое — страницы сайта — у себя на экране?

Прежде всего, на компьютере, где хранится сайт, установлена и запущена особая программа — *Web-сервер*. Она относится к весьма специфической категории программ, называемых *серверами*. Она не выводит на экран никаких окон с информацией (максимум — сообщение о критической ошибке при запуске) и вообще никак не взаимодействует с пользователем.

Единственное назначение Web-сервера — обслуживание запросов Web-обозревателей. (Которые относятся к категории программ-*клиентов*, непосредственно взаимодействующих с пользователями.)

Как только пользователь набирает в строке адреса Web-обозревателя интернет-адрес нужной ему страницы, происходит целая цепочка примечательных событий.

1. Web-обозреватель из набранного интернет-адреса извлекает адрес компьютера, на котором опубликован сайт и на котором работает программа Web-сервера, и отправляет последнему запрос, в состав которого включен весь введенный интернет-адрес целиком.
2. Web-сервер получает отправленный Web-обозревателем запрос, извлекает из находящегося в нем интернет-адреса путь к запрашиваемому файлу, считывает этот файл и отправляет назад, Web-обозревателю.

Если файл с указанным путем отсутствует или по какой-то причине не может быть прочитан, Web-сервер отправляет Web-обозревателю страницу с сообщением об ошибке, которая либо генерируется программно, либо хранится в особом файле.

3. Web-обозреватель получает отправленный Web-сервером файл и либо выводит его содержимое на экран (если это страница), либо выполняет над ним какие-то другие действия (например, предлагает сохранить его на локальном диске).

Все это происходит, так сказать, за кулисами, и пользователь не уведомляется о том, что Web-обозреватель отправляет куда-то какие-то запросы и получает откуда-то какие-то ответы. Пользователь может даже не догадываться о существовании какого-то там Web-сервера.

Интернет-адреса

А теперь подробнее рассмотрим интернет-адреса, которые набираются в строке адреса пользователем и отправляются Web-серверу Web-обозревателем.

Вообще, интернет-адрес записывается в формате:

```
[<обозначение протокола>://]<интернет-адрес сайта>[:<номер порта>]↵
[<путь к запрашиваемому файлу или папке>][?<GET-параметры>][#<имя якоря>]
```

Давайте рассмотрим все входящие в него составляющие.

- *`Обозначение протокола`* — указывает протокол и, опосредованно, серверную программу, к которой выполняется запрос.

  Протоколом называется система команд, посредством которых программа-клиент и программа-сервер взаимодействуют между собой. Будучи указанным в интернет-адресе, протокол неявно идентифицирует серверную программу, к которой отправляется запрос. Так, если указать в качестве обозначения протокола строку `http`, запрос к программе Web-сервера будет сформирован с применением протокола *HTTP* (HyperText Transfer Protocol, протокол передачи гипертекста), которым пользуются для «общения» Web-обозреватели и Web-серверы.

  Протокол HTTP будет использован и в том случае, когда обозначение протокола не указано вовсе.

- *`Интернет-адрес сайта`* — идентифицирует сам опубликованный сайт. Может представлять собой:

  - либо, как чаще всего случается, *доменное имя* — символьное наименование, удобное для запоминания и имеющее вид **www.yandex.ru**, **www.php.net** и т. п.;
  - либо низкоуровневый *IP-адрес* — четырехзначное шестнадцатеричное число вида **186.143.27.9**. Низкоуровневые IP-адреса применяются значительно реже доменных имен.

  Интернет-адрес сайта — единственная обязательная для указания часть интернет-адреса.

- `Номер порта` — указывает номер *TCP-порта*, одного из своего рода виртуальных каналов, по которым различные программы-клиенты общаются со «своими» программами-серверами, не мешая друг другу.

  Каждый тип программ-серверов по умолчанию использует один отведенный им TCP-порт — так, за Web-серверами закреплен порт 80. Однако по разным причинам сервер может быть настроен с тем, чтобы использовать другой порт, и тогда, чтобы обратиться к этому серверу, требуется явно указать номер порта в составе интернет-адреса.

  Если номер порта не указан, Web-обозреватель подразумевает, что должен применяться TCP-порт по умолчанию (для Web-сервера — порт 80).

- `Путь к запрашиваемому файлу или папке` — собственно указывает файл, который требуется получить.

  При публикации в Интернете сайта все составляющие его файлы на диске серверного компьютера помещаются в особой папке, носящей название *корневой*. Путь к этой папке указывается в настройках Web-сервера, чтобы последний смог найти ее.

  Все пути к запрашиваемым файлам, получаемые в составе интернет-адресов в запросах от Web-обозревателях, отсчитываются Web-сервером от корневой папки сайта.

  Может быть указан как путь непосредственно к запрашиваемому файлу, так и путь к папке. В последнем случае Web-сервер извлекает из этой папки и отправляет Web-обозревателю так называемый *файл по умолчанию*. Файл по умолчанию должен иметь имя index или default и расширение htm, html или php (впрочем, эти параметры можно изменить в настройках Web-сервера).

  Если путь к запрашиваемому файлу вообще не указан, будет считан и отправлен файл по умолчанию, хранящийся в корневой папке сайта. Обычно это *главная страница* сайта, с которой, собственно, и начинается «путешествие» по нему.

- `GET-параметры` — это набор данных, пересылаемых серверной программе. Мы рассмотрим их в *главе 6*.

- `Имя якоря` — указывает на фрагмент страницы. Если оно указано, содержимое страницы будет прокручено в окне Web-обозревателя таким образом, чтобы соответствующий якорю фрагмент появился в поле зрения посетителя. Если не указан, страница будет открыта как обычно. Более подробно о якорях мы поговорим в *главе 5*.

Как видим, интернет-адрес однозначно идентифицирует конкретный файл, входящий в состав конкретного сайта, и даже может указывать на конкретную часть страницы.

Вот несколько примеров интернет-адресов:

- **http://www.somesite.ru:8000/apps/app.php?page=1#chapter2**

  Доступ по протоколу HTTP к сайту **www.somesite.ru** по TCP-порту 8000. Здесь запрашивается файл серверной программы **app.php**, которая хранится в папке

apps корневой папки сайта и которой пересылается GET-параметр **page** со значением **1**, в полученной странице следует показать фрагмент, соответствующий якорю **chapter2**.

- **www.othersite.ru/folder/page1.html**

  Запрашивается файл **page1.html**, хранящийся в папке **folder** корневой папки сайта **www.othersite.ru**.

- **www.3dnews.ru**

  Запрашивается файл по умолчанию (главная страница), хранящийся в корневой папке сайта **www.3dnews.ru**.

На этом пока закончим разговор об интернет-адресах. Мы еще вернемся к ним в *главе 5*, когда будем рассматривать средства навигации по Web-сайту и, в частности, гиперссылки.

Давайте лучше приступим к изучению языка HTML, которым в дальнейшем будем пользоваться постоянно.

Введение в язык HTML 5

Содержание Web-страниц создается с применением языка *HTML* (HyperText Markup Language, язык гипертекстовой разметки). В этом разделе мы будем иметь дело исключительно с этим языком.

На данный момент самой современной его версией является HTML 5, спецификация которого совсем недавно получила статус окончательной. Ее-то мы и станем использовать для написания наших страничек.

Язык HTML довольно прост и основан на нескольких совсем не сложных принципах. Сейчас мы в этом убедимся.

Первая Web-страница

Изучать HTML лучше всего на примере. Так что давайте сразу же создадим нашу первую Web-страничку. Сделать это можно в любом простейшем текстовом редакторе — например, в Блокноте, входящем в комплект поставки Windows.

Откроем Блокнот и наберем в нем текст (или, как говорят бывалые программисты, *код*), приведенный в листинге 1.1.

Листинг 1.1

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Пример Web-страницы</title>
  </head>
```

```
  <body>
    <h1>Изучаем язык HTML</h1>
    <p>Язык <strong>HTML</strong> предназначен для создания содержимого
    Web-страниц.</p>
  </body>
</html>
```

Проверим набранный код на ошибки и сохраним в файл с именем 1.1.html. Только сделаем при этом две важные вещи.

1. Сохраним HTML-код в кодировке UTF-8. Для этого в диалоговом окне сохранения файла Блокнота найдем раскрывающийся список **Кодировка** и выберем в нем пункт **UTF-8**.
2. Заключим имя файла в кавычки. Иначе Блокнот добавит к нему расширение txt, и наш файл получит имя 1.1.html.txt.

Все, наша первая Web-страница готова! Теперь осталось открыть ее в любом Web-обозревателе (автор задействовал для этой цели Internet Explorer 11) и посмотреть на результат (рис. 1.1).

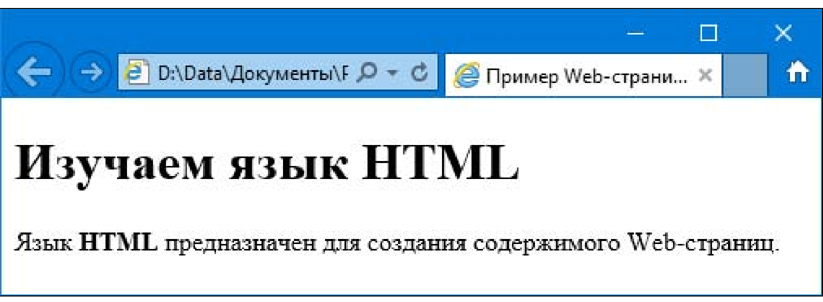


Рис. 1.1. Наша первая Web-страница

Мы только что создали Web-страницу, содержащую большой «кричащий» заголовок и абзац, включающий выделенную полужирным шрифтом аббревиатуру **HTML**. И все это — в «голом» тексте, набранном в Блокноте!

Настала пора уяснить несколько моментов, касающихся Web-страниц. Во-первых, все статические страницы представляют собой текстовые файлы. Во-вторых, файлы страниц должны иметь расширение html (используется чаще всего) или htm (на данный момент устаревшее).

Теги и атрибуты тегов

Теперь посмотрим, что же мы такое написали в файле 1.1.html. Пока что ограничимся следующим небольшим фрагментом HTML-кода:

```
<h1>Изучаем язык HTML</h1>
<p>Язык <strong>HTML</strong> предназначен для создания содержимого
Web-страниц.</p>
```

Здесь мы видим тексты заголовка и абзаца. И еще странные слова, взятые в угловые скобки, — символы < и >. Что это такое?

Это *теги* HTML, особые команды, задающие назначение того или иного фрагмента содержимого, — будет он абзацем, заголовком или важным текстом, который следует выделить полужирным шрифтом.

Начнем с тегов `<h1>` и `</h1>`, поскольку они идут первыми. Эти теги превращают фрагмент текста, находящийся между ними, в заголовок. Тег `<h1>` помечает начало фрагмента, на который распространяется действие тега, и называется *открывающим*. А тег `</h1>` устанавливает конец «охватываемого» фрагмента и называется *закрывающим*. Что касается самого фрагмента, заключенного между открывающим и закрывающим тегами, то он называется *содержимым тега*. Именно к содержимому применяется действие тега.

Все теги HTML обозначаются символами < и >, внутри которых находится *имя тега*, определяющее его назначение (имена тегов можно набирать как строчными, так и прописными буквами, но в HTML 5 обычно используются строчные буквы). Закрывающий тег должен иметь то же имя, что и открывающий, единственное отличие закрывающего тега — символ `/`, который ставится между символом < и именем тега.

Рассмотренные нами теги `<h1>` и `</h1>` в HTML фактически считаются одним тегом `<h1>`. Такой тег называется *парным*.

Другой парный тег — `<p>` — создает на Web-странице абзац из текста, являющегося содержимым этого тега. Такой абзац будет отображаться с отступами сверху и снизу. Если он полностью помещается по ширине в окне Web-обозревателя, то отобразится в одну строку, в противном случае Web-обозреватель сам выполнит перенос строк по пробелам (это же справедливо и для заголовка).

Третий парный тег — `<strong>` — помечает свое содержимое как важный текст, на котором следует заострить внимание посетителя, для чего выводит его полужирным шрифтом. Видно, что этот тег вложен внутрь содержимого тега `<p>`. Это значит, что содержимое тега `<strong>` будет отображаться как часть абзаца (тега `<p>`).

А теперь рассмотрим вот такой тег:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Это *одинарный тег*, не имеющий закрывающей пары. Такие теги действует в той точке HTML-кода, где они находятся, и либо задают какие-либо сведения о самой странице (как приведенный здесь тег `<meta>`), либо помещают в соответствующее место страницы какой-либо элемент, не относящийся к тексту.

А еще мы видим, что в этом теге сразу после имени тега идут еще какие-то данные. Это *атрибуты тега*, задающие его параметры. В частности, атрибуты `http-equiv` и `content` тега `<meta>` указывают тип документа и его кодировку.

Каждый атрибут тега имеет *имя*, за которым ставится знак равенства и *значение* этого атрибута, взятое в двойные кавычки. Так, атрибут с именем `http-equiv` имеет значение `"Content-Type"`, указывающее, что этот тег задает тип документа. А атри-

бут с именем content имеет значение "text/html; charset=utf-8", обозначает тип документа «Web-страница» и указывает, что он сохранен в кодировке UTF-8.

Атрибуты тегов бывают обязательными и необязательными. *Обязательные* атрибуты должны присутствовать в теге в обязательном порядке. *Необязательные* же атрибуты могут быть опущены — в таком случае тег ведет себя так, будто соответствующему атрибуту присвоено значение по умолчанию. Атрибуты http-equiv и content тега <meta> являются обязательными.

Вложенность тегов

Если мы снова посмотрим на приведенный в листинге 1.1 HTML-код, то заметим, что одни теги вложены в другие. Так, тег <strong> вложен в тег <p>, являясь частью его содержимого. Тег <p>, в свою очередь, вложен в тег <body>, а тот — в тег <html>. (Теги <body> и <html> мы рассмотрим чуть позже.) Такая *вложенность тегов* в HTML — обычное явление.

Когда Web-обозреватель встречает тег, вложенный в другой тег, он как бы накладывает действие «внутреннего» тега на эффект «внешнего». Так, действие тега <strong> будет наложено на действие тега <p>, и фрагмент абзаца окажется выделенным полужирным шрифтом, при этом оставаясь частью этого абзаца.

Теперь — внимание! Порядок следования закрывающих тегов должен быть обратным тому, в котором следуют теги открывающие. Говоря иначе, теги со всем их содержимым должны полностью вкладываться в другие теги, не оставляя «хвостов» снаружи.

Осталось выучить несколько новых терминов. Тег, в который непосредственно вложен тот или иной тег, называется *родительским*, или *родителем*. В свою очередь, тег, вложенный в родительский тег, называется *дочерним*, или *потомком*. Так, для тега <p> в приведенном ранее листинге тег <body> — родительский, а тег <strong> — дочерний. Любой тег может иметь сколько угодно дочерних тегов, но только один родительский (что, впрочем, понятно — не может же он быть непосредственно вложен одновременно в два тега).

Аналогично, элемент Web-страницы, в который вложен элемент, создаваемый каким-либо тегом, называется *родительским*, или *родителем*. А элемент страницы, который вложен в этот элемент, — *дочерним*, или *потомком*.

Уровень вложенности того или иного тега показывает количество тегов, в которые он последовательно вложен. Так, если принять за точку отсчета тег <html>, то тег <body> будет иметь первый уровень вложенности, т. к. он вложен непосредственно в тег <html>. Тег <p> же будет иметь второй уровень вложенности, т. к. он вложен в тег <body>, а тот, в свою очередь, — в тег <html>. В сложных страницах уровень вложенности иных тегов может составлять несколько десятков.

Форматирование Web-страниц

А теперь рассмотрим несколько важных моментов, касающихся структуры HTML-кода страницы и сведений о ней, что необходимы Web-обозревателю для успешного вывода ее содержания.

Секции Web-страницы

Прежде всего, содержание любой страницы разделяется на две *секции*. Для этого применяются особые теги, которые часто называют *невидимыми*, поскольку они никак не отображаются на экране, по крайней мере, напрямую.

Давайте мысленно удалим из листинга 1.1 все содержимое, за исключением этих тегов, в результате чего получим листинг 1.2.

Листинг 1.2

```
<html>
  <head>
    . . .
  </head>
  <body>
    . . .
  </body>
</html>
```

Начнем с парного тега `<body>`, о котором уже упоминали ранее. Он формирует *секцию тела* страницы, которая описывает собственно содержание страницы, выводящееся на экран. Все теги, что формируют содержание, должны находиться в этой секции:

```
<body>
  <h1>Изучаем язык HTML</h1>
  <p>Язык <strong>HTML</strong> предназначен для создания содержимого
  Web-страниц.</p>
</body>
```

А в парном теге `<head>` находится *секция заголовка* Web-страницы. (Не путать с заголовком, который создается с помощью тега `<h1>`!) В эту секцию помещаются сведения о параметрах страницы, не отображаемые на экране и предназначенные исключительно для Web-обозревателя (например, знакомый нам тег `<meta>`):

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>Пример Web-страницы</title>
</head>
```

И заголовок, и тело страницы находятся внутри парного тега `<html>`. Этот тег расположен на самом высшем (нулевом) уровне вложенности и не имеет родителя.

Любая страница должна быть правильно отформатирована: иметь секции заголовка и тела и необходимый набор метатегов (о которых мы также поговорим). Только в этом случае она будет считаться корректной с точки зрения стандартов HTML.

Метаданные и метатеги

Мы только что узнали о секции заголовка страницы и о том, что она предназначена для задания параметров страницы. Эти параметры называются *метаданными*, т. е. данными, описывающими другие данные. А для их задания применяются *метатеги*, относящиеся к числу невидимых тегов.

Прежде всего, в состав метаданных входит *название* Web-страницы. Оно отображается в заголовке окна Web-обозревателя, где выводится содержание этой страницы, и хранится в «истории» (списке посещенных к настоящему времени страниц). Название Web-страницы помещается в парный тег `<title>`:

```
<head>
  . . .
  <title>Пример Web-страницы</title>
</head>
```

Далее, это уже знакомый нам тег `<meta>`, задающий тип документа и кодировку текста, которым он набран.

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  . . .
</head>
```

Приведенный тег указывает, что наш документ представляет собой Web-страницу, и задает для него кодировку текста UTF-8.

> ***НА ЗАМЕТКУ***
>
> Кодировка *UTF-8* — это разновидность кодировки Unicode, предназначенная для написания Web-страниц. Кодировка Unicode (а значит, и UTF-8) может закодировать все символы всех языков, имеющихся на Земле.

> ***ВНИМАНИЕ!***
>
> Для кодирования текста страниц, написанных на языке HTML 5, настоятельно рекомендуется применять кодировку UTF-8. Использование других кодировок хоть и допускается, но не приветствуется.

Теперь осталось рассмотреть последний тег, находящийся в самом начале HTML-кода нашей страницы — даже вне «всеобъемлющего» тега `<html>`. Это метатег `<!doctype>`, который задает, во-первых, версию языка HTML, на которой написана страница, а во-вторых, разновидность этой версии.

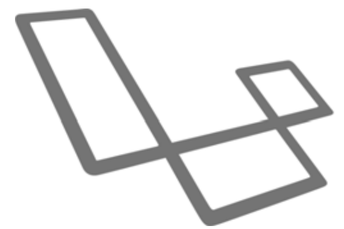
В нашем случае тег `<!doctype>` выглядит так:

```
<!doctype html>
```

Он указывает, что для создания страницы применяется версия 5 языка HTML.

Внимание!

Код любой страницы, написанной на HTML 5, должен включать метатег `<!doctype html>`, расположенный в самом его начале. В противном случае Web-обозреватель не сможет правильно обработать страницу.



ГЛАВА 2

Структурирование и оформление текста. Литералы. Комментарии HTML

В предыдущей главе мы изучили основы языка HTML, на котором создается содержание страниц. В этой главе мы познакомимся с тегами, с помощью которых содержание страниц структурируется и оформляется.

Структурирование текста

Сначала мы изучим средства HTML по структурированию текста — разбиению его на абзацы, заголовки, списки и т. п. Это выполняется с помощью особых тегов.

И начнем мы с самых простых и самых «старых» средств, появившихся еще в первых версиях HTML.

Абзацы и заголовки

Из чего состоит текст? Правильно, из отдельных абзацев, включающих логически законченные и относительно независимые фрагменты текста, и заголовков, дающих названия отдельным частям текста: разделам, главам и параграфам.

Как мы уже знаем из *главы 1*, для создания абзаца применяется парный тег `<p>`. Содержимое этого тега становится текстом абзаца:

```
<p>Я - совсем короткий абзац.</p>
<p>А я - уже более длинный абзац. Возможно, Web-обозреватель разобьет
меня на две строки.</p>
```

Теперь о заголовках. Скажем сразу, что в HTML есть такое понятие, как *уровень заголовка*. Он представляет собой целое число от 1 до 6, указывающее, насколько крупную часть текста открывает заголовок.

- Заголовок первого уровня (1) открывает самую крупную часть текста. Как правило, это заголовок всей Web-страницы. Web-обозреватель выводит заголовок первого уровня самым большим шрифтом.

- Заголовок второго уровня (2) открывает более мелкую часть текста. Обычно это большой раздел. Web-обозреватель выводит заголовок второго уровня несколько меньшим шрифтом, чем заголовок первого уровня.
- Заголовок третьего уровня (3) открывает еще более мелкую часть текста — обычно главу в разделе. Web-обозреватель выводит такой заголовок еще меньшим шрифтом.
- Заголовки четвертого, пятого и шестого уровней (4–6) открывают отдельные параграфы, крупные, более мелкие и самые мелкие соответственно. Web-обозреватель выводит заголовки четвертого уровня еще меньшим шрифтом, а шрифт заголовков пятого и шестого уровней — даже меньше, что шрифт, которым выводятся обычные абзацы.

Чтобы дополнительно выделить заголовки любого уровня, Web-обозреватель выводит их полужирным шрифтом.

Если мы откроем в Блокноте нашу первую страницу, сохраненную в файле 1.1.html, заменим в ней содержимое секции тела (тега <body>) на код, приведенный в листинге 2.1, сохраним под именем 2.1.html и откроем в Web-обозревателе, мы увидим то, что показано на рис. 2.1. Так мы можем примерно оценить размер шрифта, которым выводятся заголовки различных уровней.

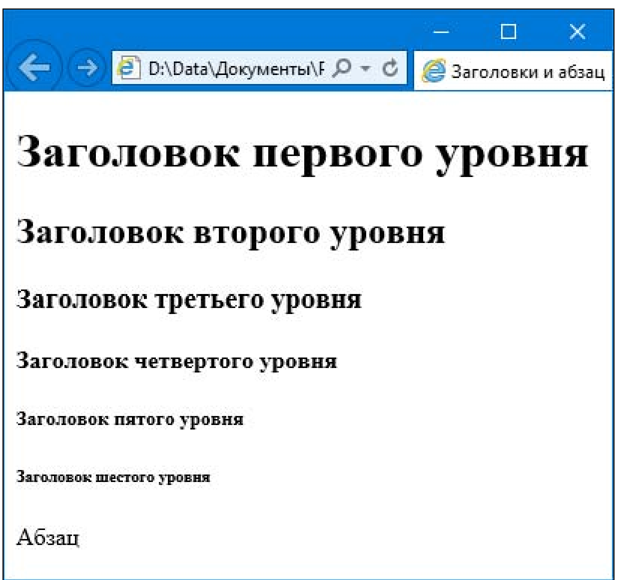


Рис. 2.1. Заголовки различных уровней и абзац

Листинг 2.1

```
<h1>Заголовок первого уровня</h1>
<h2>Заголовок второго уровня</h2>
<h3>Заголовок третьего уровня</h3>
```

```
<h4>Заголовок четвертого уровня</h4>
<h5>Заголовок пятого уровня</h5>
<h6>Заголовок шестого уровня</h6>
<p>Абзац</p>
```

Блочные элементы HTML

А теперь рассмотрим один важный вопрос. Он касается типов элементов страниц, поддерживаемых HTML, и правил, согласно которым они выводятся на экран.

Уясним сразу, что абзацы и заголовки — это так называемые *блочные элементы*. Такие элементы:

- выводятся по вертикали в направлении сверху вниз;
- часто отделяются отступами от соседних блочных элементов;
- занимают все свободное пространство по ширине;
- принимают такую высоту, чтобы только вместить свое содержимое;
- могут содержать как обычный текст, так и другие блочные элементы;
- в случае необходимости Web-обозреватель сам выполняет перенос текста — содержимого блочных элементов.

И, если уж зашла речь о текстовом содержимом тегов, давайте приведем правила, согласно которым выполняется его вывод:

- два и более следующих друг за другом пробела считаются за один пробел;
- табуляция и перевод строки считается за пробел;
- пробелы, табуляции и переводы строки между тегами, создающие блочные элементы, никак не отображаются на Web-странице. (Благодаря этому мы можем форматировать HTML-код для более удобного чтения, в том числе, ставить отступы для обозначения вложенности тегов.)

> ***Внимание!***
>
> Все без исключения теги, предназначенные для структурирования текста, создают блочные элементы.

Списки

Списки используются для того, чтобы представить читателю перечень каких-либо позиций, пронумерованных или непронумерованных, — пунктов списка. Список с пронумерованными пунктами так и называется — *нумерованным*, а с непронумерованными — *маркированным*. В маркированных списках пункты помечаются особым значком — *маркером*, который ставится левее пункта списка.

Маркированные списки обычно служат для простого последовательного вывода каких-либо позиций, порядок следования которых не важен. Если же требуется обратить внимание читателя на то, что позиции должны следовать друг за другом

именно в том порядке, в котором они перечислены, необходимо применить нумерованный список.

Любой список в HTML создается в два этапа. Сначала пишут строки, которые станут пунктами списка, и каждую из этих строк помещают внутрь парного тега <li>. Затем все эти пункты помещают внутрь парного тега <ul> (если создается маркированный список) или <ol> (в случае создания нумерованного списка) — эти теги определяют сам список (листинг 2.2).

Листинг 2.2

```
<ul>
  <li>Я - первый пункт маркированного списка.</li>
  <li>Я - второй пункт маркированного списка.</li>
  <li>Я - третий пункт маркированного списка.</li>
</ul>
<ol>
  <li>Я - первый пункт нумерованного списка.</li>
  <li>Я - второй пункт нумерованного списка.</li>
  <li>Я - третий пункт нумерованного списка.</li>
</ol>
```

Web-обозреватель сам расставляет в пунктах списков необходимые маркеры или нумерацию. Также он выводит пункты списка с отступом слева, а расстояние между ними делает меньшими, чем в случае абзацев или заголовков.

Списки можно помещать друг в друга, создавая *вложенные списки*. Делается это следующим образом. Сначала во «внешнем» списке (в который должен быть помещен вложенный) отыскивают пункт, после которого должен находиться вложенный список. Затем HTML-код, создающий вложенный список, помещают в разрыв между текстом этого пункта и его закрывающим тегом . Если же вложенный список должен помещаться в начале «внешнего» списка, его следует вставить между открывающим тегом <li> первого пункта «внешнего» списка и его текстом. Что, впрочем, логично.

В листинге 2.3 представлен HTML-код, создающий два списка, один из которых вложен внутрь другого. Обратим внимание, где помещается HTML-код, создающий вложенный список.

Листинг 2.3

```
<ul>
  <li>Я - первый пункт внешнего списка.</li>
  <li>Я - второй пункт внешнего списка.
    <ul>
      <li>Я - первый пункт вложенного списка.</li>
      <li>Я - второй пункт вложенного списка.</li>
      <li>Я - третий пункт вложенного списка.</li>
```

```
    </ul>
  </li>
  <li>Я - третий пункт внешнего списка.</li>
</ul>
```

HTML позволяет вкладывать нумерованный список внутрь маркированного и наоборот. Количество последовательно вложенных друг в друга списков не ограничено.

HTML также позволяет создать так называемый *список определений*, представляющий собой перечень терминов и их разъяснений. Такой список создают с помощью парного тега `<dl>`. Внутри него помещают пары «термин — разъяснение». Термины заключают в парный тег `<dt>`, а разъяснения — в парный тег `<dd>` (листинг 2.4).

Листинг 2.4

```
<dl>
  <dt>Содержание</dt>
  <dd>Информация, отображаемая на Web-странице</dd>
  <dt>Представление</dt>
  <dd>Набор правил, определяющих формат отображения содержания</dd>
  <dt>Поведение</dt>
  <dd>Набор правил, определяющих реакцию Web-страницы или ее элементов на
  действия посетителя</dd>
</dl>
```

Блочные цитаты и адреса

HTML предлагает нам еще два полезных тега. Первый позволяет создать большую цитату, включающую несколько абзацев, а второй — указать адрес, которым могут быть контактные данные разработчиков сайта, дата написания статьи или просто подпись ее автора.

Блочная цитата создается парным тегом `<blockquote>`. В нем помещается HTML-код, собственно формирующий блочную цитату (листинг 2.5).

Листинг 2.5

```
<blockquote>
  <p>Я - начало блочной цитаты.</p>
  <p>Я - продолжение блочной цитаты.</p>
</blockquote>
```

Как видим, в тег `<blockquote>` можно поместить несколько абзацев. Там также могут иметься заголовки и списки (если уж возникнет такая потребность).

Блочная цитата выводится на экран с отступом слева.

Что касается адреса, то он создается парным тегом `<address>`. Он ведет себя так же, как тег абзаца `<p>`, но его содержимое выводится курсивом:

```
<address>Я - адрес создателя этой Web-страницы: почтовый, электронный,
телефоны и факсы.</address>
```

Текст фиксированного формата

Ранее мы ознакомились с правилами, согласно которым Web-обозреватель выполняет вывод текста. Эти правила, в частности, гласят, что несколько стоящих подряд пробелов считаются за один пробел, также за пробел считаются переводы строк и табуляция.

Однако часто бывает необходимо вывести какой-либо фрагмент текста как есть, со всеми множественными пробелами, стоящими подряд, и переводами строк. К таким фрагментам можно отнести, например, исходные тексты программ, — они набираются согласно правилам, определяемым языком программирования, на которых написаны, и, следовательно, выводиться они должны точно так же, как и набраны.

Специально для таких случаев HTML предусматривает парный тег `<pre>`. Внутри него помещается текст, который должен быть выведен так же, как он набран, или, как говорят Web-верстальщики, текст *фиксированного формата*.

Листинг 2.6 демонстрирует пример кода, выводящего текст фиксированного формата, а на рис. 2.2 показано, как этот текст выглядит на экране.

Листинг 2.6

```
<pre>Этот текст
  будет выведен
    на Web-страницу
      как есть,
        без всяких преобразований.</pre>
```

Правила отображения текста фиксированного формата:

- для вывода используется моноширинный, а не пропорциональный шрифт;
- все пробелы, табуляция и переносы строк сохраняются при выводе (это мы уже знаем);

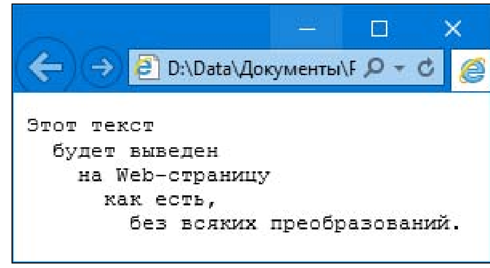


Рис. 2.2. Текст фиксированного формата

- если строка текста фиксированного формата не помещается в окне Web-обозревателя по ширине, она ни в коем случае не будет переноситься. Из-за этого может возникнуть потребность в горизонтальной прокрутке Web-страницы;
- допускается оформлять фрагмент текста фиксированного формата и создавать в нем гиперссылки, используя соответствующие теги (о которых будет рассказано далее в этой главе и в *главе 5*).

Блочные контейнеры

Иногда бывает необходимо объединить несколько блочных элементов: абзацев, заголовков, списков — в один элемент. Это может потребоваться при создании разметки страницы, интерактивного элемента или просто чтобы привязать к этому элементу какой-либо стиль (о стилях речь пойдет в *разделе 2*).

Для таких случаев HTML предусматривает парный тег `<div>`. Внутри него помещается HTML-код, создающий содержимое, которое должно быть объединено в сущность, называемую *блочным контейнером*, или *блоком*:

```
<div>
  <p>Я вхожу в состав блока.</p>
  <p>Я тоже вхожу в состав блока.</p>
</div>
```

Web-обозреватели при выводе никак не выделяют блочные контейнеры. Однако мы можем, как было сказано ранее, привязать к ним стиль, который задаст для них нужное нам оформление.

Блоки очень пригодятся нам в дальнейшем, когда мы начнем работать со стилями, создавать разметку и интерактивные элементы страниц.

Семантическая разметка текста

Описанных ранее простейших инструментов для структурирования текста, предлагаемых HTML, хватает во многих случаях. Но если страница содержит большой текст, разбитый на множество частей, их может оказаться недостаточно. Поэтому HTML 5 предоставляет в наше распоряжение набор новых тегов, выполняющих так называемую *семантическую разметку* текста.

Семантическая разметка заключается в разбиении текста на отдельные значащие блоки. Такими блоками могут быть сама статья, ее «шапка» или «поддон», отдельная часть статьи, примечание, иллюстрация в виде рисунка с подписью, набор гиперссылок для навигации и др.

Давайте рассмотрим все теги, используемые для семантической разметки, с указанием области их применения. Все они являются парными.

- `<article>` — независимый и самодостаточный фрагмент основного содержания страницы: статья, отдельный ее фрагмент, отдельная запись на форуме, в блоге или отдельный комментарий.

- `<header>` — «шапка» статьи или всей страницы.
- `<footer>` — «поддон» статьи или всей страницы.
- `<section>` — значимая часть материала, например, параграф большой статьи. Также может использоваться в качестве составной части разметки страницы (см. *главу 13*).

```
<article>
  ...Большая статья...
  <header>
    ...Заголовок...
    ...Имя автора...
    ...Оглавление...
  </header>
  <section>
    ...Первый параграф...
  </section>
  <section>
    ...Второй параграф...
  </section>
  . . .
  <footer>
    ...Предметный указатель...
    ...Список сопутствующих материалов...
    ...Дата публикации...
  </footer>
</article>
```

- `<nav>` — набор гиперссылок, предназначенных для навигации по сайту, или целая панель навигации (см. *главу 5*).
- `<aside>` — примечание к статье, обычно располагающееся сбоку от основного текста.

```
<article>
  ...Статья...
  . . .
  <aside>
    ...Примечание...
  <aside>
  . . .
</article>
```

  Может использоваться для формирования всевозможных плавающих элементов (см. *главу 13*).

- `<figure>` — иллюстрация к статье: обычное графическое изображение, аудио-, видеоролик или фрагмент текста.
- `<figcaption>` — подпись к иллюстрации. Может присутствовать только в теге `<figure>`.

```
<article>
  ...Статья...
  . . .
  <figure>
    <img src="/images/screen1.png">
    <figcaption>
      <P>Скриншот 1</P>
    </figcaption>
  </figure>
  . . .
</article>
```

- `<mark>` — важный фрагмент, на который следует обратить внимание посетителя. Выделяется желтым фоном.

На данный момент Web-обозреватели никак не выделяют визуально теги семантической разметки. (Единственное исключение — важный фрагмент, создаваемый тегом `<mark>`, содержимое которого выводится на желтом фоне.) Однако мы всегда сможем привязать к ним стили, чтобы задать нужное нам оформление.

Горизонтальные линии

Горизонтальная линия в HTML иногда применяется для визуального отделения одного фрагмента текста от другого. Она создается с помощью одинарного тега `<hr>`:

```
<p>Я отделен от следующего абзаца горизонтальной линией.</p>
<hr>
<p>Я отделен от предыдущего абзаца горизонтальной линией.</p>
```

Горизонтальная линия HTML растягивается на все свободное пространство по ширине, имеет один-два пиксела в толщину и выпуклый или вдавленный вид (конкретные параметры зависят от Web-обозревателя).

> ***На заметку***
>
> В настоящее время применение горизонтальных линий HTML считается дурным тоном Web-дизайна. Вместо них рекомендуется применять рамки, создаваемые средствами CSS (подробнее — в *главе 12*).

Оформление текста

На этом разговор о структурировании текста средствами HTML закончен. Поговорим об оформлении текста.

Выделение фрагментов текста

Собственно, средства HTML для оформления текста мы начали изучать еще в *главе 1*. Это парный тег `<strong>`, который помечает свое содержимое как очень важный текст и для привлечения к нему внимания выводит его полужирным шрифтом.

HTML предусматривает для выделения текста довольно много тегов (табл. 2.1), имеющих две особенности: все они парные и служат для выделения частей текста блочных элементов.

Таблица 2.1. *Теги HTML, предназначенные для выделения фрагментов текста*

<table>
<tr><th>Тег</th><th>Назначение</th><th>Выделяется Web-обозревателем</th></tr>
<tr><td><strong></td><td>Очень важный текст</td><td>Полужирным шрифтом</td></tr>
<tr><td><em></td><td>Важный текст</td><td>Курсивом</td></tr>
<tr><td><abbr></td><td>Аббревиатура</td><td></td></tr>
<tr><td><dfn></td><td>Первое появление в тексте нового термина</td><td>Курсивом</td></tr>
<tr><td><q></td><td>Небольшая цитата</td><td>Кавычками</td></tr>
<tr><td><cite></td><td>Заголовок книги, статьи, название картины, фильма и т. п.</td><td>Курсивом</td></tr>
<tr><td><code></td><td>Небольшой фрагмент исходного кода программы</td><td>Моноширинным шрифтом</td></tr>
<tr><td><var></td><td>Имя переменной в исходном коде программы</td><td>Курсивом</td></tr>
<tr><td><kbd></td><td>Данные, вводимые пользователем в какую-либо программу</td><td rowspan="2">Моноширинным шрифтом</td></tr>
<tr><td><samp></td><td>Данные, выводимые какой-либо программой</td></tr>
<tr><td><ins></td><td>Текст, вновь помещенный на страницу</td><td>Подчеркиванием</td></tr>
<tr><td><del></td><td>Удаленный текст</td><td rowspan="2">Зачеркиванием</td></tr>
<tr><td><s></td><td>Текст, нуждающийся в проверке</td></tr>
</table>

Рассмотрим несколько примеров использования этих тегов (листинг 2.7).

Листинг 2.7

```
<p>Язык <dfn>HTML</dfn> служит для создания <em>содержимого
Web-страниц</em>.</p>
<p><strong>Соблюдайте порядок вложенности тегов!</strong></p>
<p>Тег <code>p</code> служит для создания абзаца HTML.</p>
<p>Наберите в Web-обозревателе интернет-адрес
<kbd>http://www.w3c.org</kbd>.</p>
```

Из всех рассмотренных нами тегов чаще всего встречаются `<strong>` и `<em>`. Остальные теги так и не снискали большой популярности среди разработчиков.

Встроенные элементы HTML

Мы только что рассмотрели ряд тегов, с помощью которых можно каким-либо образом выделить фрагмент текста. Это весьма примечательные теги, о которых следует сказать еще пару слов.

Как уже говорилось ранее, все они служат для выделения фрагментов текста, являющихся частью блочных элементов, скажем, абзацев. Элементы Web-страницы, которые они создают, называются *встроенными*.

Встроенные элементы:

- выводятся по горизонтали, в направлении слева направо, как и символы текста;
- выводятся вплотную друг к другу — по горизонтали и на расстоянии межстрочного интервала — по вертикали;
- принимают такие размеры, чтобы только вместить свое содержимое;
- могут находиться только внутри блочных элементов;
- могут содержать как обычный текст, так и другие встроенные элементы. Блочные элементы они содержать не могут.

Встроенные контейнеры

Ранее в этой главе мы изучили блочные контейнеры, объединяющие несколько блочных элементов и также представляющие собой блочные элементы.

HTML позволяет создавать и *встроенные контейнеры*, включающие в себя фрагмент содержимого блочного элемента, — например, с целью привязать к этому фрагменту некий стиль. Понятно, что такие контейнеры сами являются встроенными элементами.

Для создания встроенного контейнера применяется парный тег `<span>`:

```
<p><span>Этот фрагмент</span> является содержимым встроенного
контейнера.</p>
```

Разрыв строки

Если строка — содержимое какого-либо блочного элемента слишком длинна и не помещается в отведенное ей пространство целиком, Web-обозреватель сам выполнит ее перенос. Это мы уже знаем.

Но HTML позволяет нам выполнить перенос в произвольном месте строки. Причем в этом случае строка будет переноситься всегда.

Как это сделать? Просто задать так называемый *разрыв строки*, вставив в нужном ее месте одинарный тег `<br>`:

```
<p>Этот абзац будет разорван на две строки в этом<br>месте.</p>
```

Разрыв строки также относится к встроенным элементам страницы. Его можно применять для разбиения абзаца на несколько более или менее независимых частей, которые, тем не менее, относятся к одной и той же теме.

Вставка специальных символов. Литералы

Язык HTML использует определенные символы, в частности, знаки <, > и двойные кавычки, для написания тегов и их атрибутов. Если вставить любой из этих символов в обычный текст (в содержимое тега), Web-обозреватель не только не выведет этот символ на экран, но и, скорее всего, не сможет правильно обработать фрагмент текста, в котором он находится.

Для вставки в текст подобных символов (они называются *специальными*) служат особые последовательности знаков, называемые *литералами*. Встретив литерал, Web-обозреватель «поймет», что здесь должен присутствовать соответствующий специальный символ, и выведет его на Web-страницу.

Литералов в HTML довольно много. Наиболее часто применяемые из них приведены в табл. 2.2.

Таблица 2.2. *Некоторые литералы языка HTML*

Специальный символ	**Литерал HTML**
Неразрывный пробел	` `
" (обычная двойная кавычка)	`"`
<	`<`
>	`>`
&	`&`
©	`©`
®	`®`
— (длинное тире)	`—`
– (короткое тире)	`–`
… (многоточие)	`…`
Левая двойная кавычка	`“`
Правая двойная кавычка	`”`
Левая угловая кавычка	`«`
Правая угловая кавычка	`»`
Левый апостроф	`‘`
Правый апостроф	`’`
Символ евро	`€`

Полный список поддерживаемых языком HTML литералов можно найти по интернет-адресу **http://htmlbook.ru/samhtml/tekst/spetssimvoly**.

Пара примеров:

```
<p>Тег &lt;body&gt; формирует секцию тела страницы.</p>
<address>&copy; Владимир Дронов.</address>
```

Среди приведенных в табл. 2.2 литералов и обозначаемых ими специальных символов особенно выделяется один. Это *неразрывный пробел*, обозначаемый литералом ` `. По этому пробелу Web-обозреватель никогда не будет выполнять перенос строк.

Неразрывный пробел необходим, если в каком-то месте предложения перенос строк не должен выполняться в любом случае. Так, правила правописания русского языка не допускают перенос строк перед длинным тире. Поэтому настоятельно рекомендуется отделять длинное тире от предыдущего слова неразрывным пробелом:

```
<p>Неразрывный пробел &mdash; очень важный литерал.<p>
```

HTML также позволяет вставить в текст любой символ, поддерживаемый кодировкой Unicode, просто указав его код. Для этого предусмотрен литерал вида &#*<десятичный код символа>*;.

Но как узнать код нужного символа? Очень просто. В этом нам поможет утилита Таблица символов, поставляемая в составе Windows. Давайте запустим ее и посмотрим на ее окно (рис. 2.3).

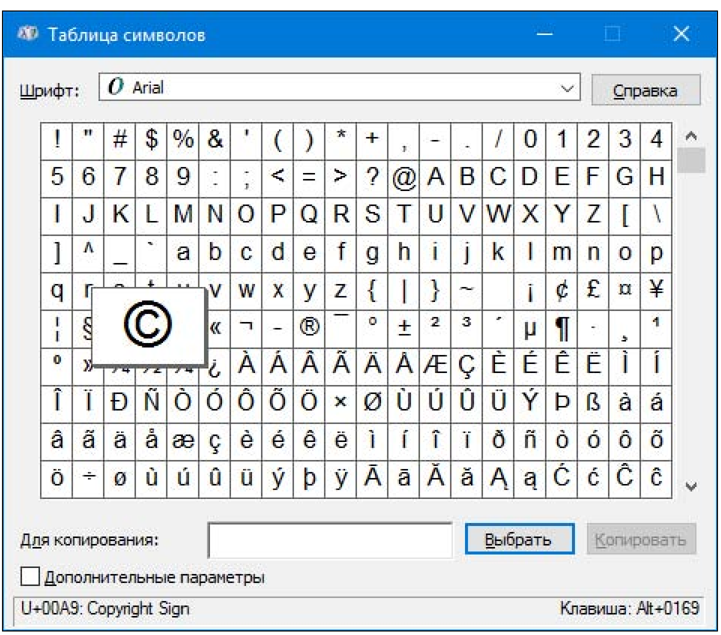


Рис. 2.3. Окно утилиты Таблица символов (выбран символ ©)

В большом списке символов, занимающем почти все окно этой утилиты, выберем нужный нам символ. После этого посмотрим на строку статуса, расположенную вдоль нижнего края окна. В правой ее части находится надпись вида **Клавиша: Alt+<*десятичный код символа*>**. Этот-то код нам и нужен!

> ***Внимание!***
>
> Надпись **Клавиша: Alt+<*десятичный код символа*>** появляется в строке статуса окна Таблицы символов только при выборе символов, которые нельзя ввести непосредственно с клавиатуры.

Так, мы можем вставить в сведения об авторских правах символ ©, использовав литерал `©`, где `0169` — десятичный код этого символа (см. рис. 2.3):

```
<address>&#0169; Владимир Дронов.</address>
```

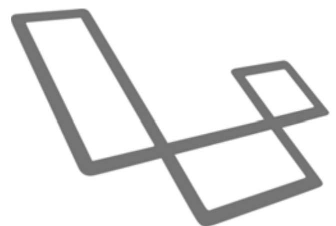
Комментарии

Напоследок рассмотрим одну очень важную возможность HTML, которая, хоть и не касается напрямую Web-верстки, но очень поможет забывчивым Web-верстальщикам.

Комментарий — это фрагмент HTML-кода, который не выводится на экран и вообще не обрабатывается Web-обозревателем. Он служит для того, чтобы разработчик страницы смог оставить текстовые заметки для себя или своих коллег.

Текст комментария HTML помещают между открывающим тегом `<!--` и закрывающим тегом `-->` и обязательно отделяют от этих тегов пробелами. Как видим, в этом случае открывающий и закрывающий теги — разные.

```
<!-- Я - комментарий. Меня не видно на Web-странице. -->
<!-- Здесь должна быть панель навигации.
Не забыть добавить ее. -->
```



ГЛАВА 3

Графика и мультимедиа

Web-страницы могут содержать не только текст. Возможность вывода графических изображений присутствовала уже в самой первой версии языка HTML. А HTML 5 позволяет нам без малейшего труда разместить на страницах аудио- и видеоролики, причем, не прибегая к помощи сторонних программ, таких как Adobe Flash Player.

Так что совершенно не удивительно, что уже третью главу этой книги автор посвятил выводу на страницах графики и мультимедийных данных.

Внедренные элементы Web-страниц

Казалось бы, что может быть проще: описывай прямо в HTML-коде графическое изображение, аудио- или видеоклип — и все будет замечательно! Но не тут-то было... Все дело в том, что графика и мультимедийные данные имеют принципиально иную природу, нежели текст. Из-за этого объединить их в одном файле невозможно.

Разработчики HTML нашли оригинальный выход из положения. Прежде всего, графические изображения и мультимедийные данные сохраняются в отдельных файлах. А в HTML-коде страниц с помощью особых тегов записываются интернет-адреса этих файлов. Встретив такой тег-ссылку, Web-обозреватель запрашивает у Web-сервера соответствующий файл с изображением, аудио- или видеороликом и выводит его на страницу в то место, где встретился этот тег.

Графические изображения, аудио- и видеоролики и вообще любые данные, хранящиеся в отдельных от страниц файлах, называются *внедренными* элементами Web-страниц.

Графика

Как уже говорилось, графические изображения — суть внедренные элементы страниц. Это значит, что они сохраняются в отдельных от самой страницы файлах.

Поддерживаемые форматы интернет-графики

В настоящее время существует несколько десятков форматов хранения графики в файлах. Но Web-обозреватели поддерживают далеко не все. В WWW сейчас используются всего четыре формата, которые мы далее рассмотрим.

- Формат *GIF* (Graphics Interchange Format, формат обмена графикой) — старожил среди «сетевых» форматов графики. Он был разработан еще в 1987 году и модернизирован в 1989 году. Сейчас он считается устаревшим, но все еще широко распространен.

  Достоинств у него довольно много. Во-первых, GIF позволяет задать для изображения «прозрачный» цвет. Закрашенные этим цветом области изображения станут своего рода «дырками», сквозь которые будет «просвечивать» фон родительского элемента, — это позволяет накладывать одно изображение на другое. Во-вторых, в одном GIF-файле можно хранить сразу несколько изображений, фактически — настоящий фильм (*анимированный GIF*). В-третьих, из-за особенностей применяемого в этом формате алгоритма сжатия он отлично подходит для хранения штриховых изображений (карт, схем, рисунков карандашом и проч.).

  Недостаток у формата GIF всего один — он совершенно не годится для хранения полутоновых изображений (фотографий, картин и т. п.). Это обусловлено тем, что GIF-изображения могут включать всего 256 цветов, и потерями качества при сжатии.

  GIF используется для хранения элементов оформления Web-страниц (всяческих линеек, маркеров списков и т. п.), штриховых изображений и несложной анимации.

- Формат *JPEG* (Joint Photographic Experts Group, объединенная группа экспертов по фотографии) был разработан в 1993 году специально для хранения полутоновых изображений. Для чего активно применяется до сих пор.

  JPEG, в отличие от GIF, не ограничивает количество цветов у изображения, а реализованный в нем алгоритм сжатия лучше всего подходит для полутоновых изображений. Однако он плохо справляется с штриховой графикой, не поддерживает «прозрачный» цвет и анимацию.

- Формат *PNG* (Portable Network Graphics, перемещаемая сетевая графика) появился на свет в 1996 году. Он разрабатывался как замена устаревшему и не очень удобному GIF, а также, в некоторой степени, JPEG. В настоящее время он практически вытеснил GIF и постепенно отвоевывает «жизненное пространство» у JPEG.

  К достоинствам формата PNG можно отнести возможность хранения как штриховых, так и полутоновых изображений и поддержку полупрозрачности. Недостаток всего один и некритичный — отсутствие поддержки анимации.

- Первая версия формата *SVG* (Scalable Vector Graphics, масштабируемая векторная графика) была представлена в 2001 году, а в 2011-м вышла версия 1.1, акту-

альная на данный момент. Сейчас этот формат медленно, но верно набирает популярность.

Преимущества формата SVG: хранение изображений в текстовых файлах в виде описания набора фигур, из которых оно состоит, вследствие чего изображение может быть, при наличии достаточного усердия, создано в обычном текстовом редакторе; масштабирование без потерь качества; поддержка анимации; полная открытость (разработчикам приложений с поддержкой SVG не придется платить лицензионных отчислений). Ключевой недостаток, пожалуй, всего один — большой размер файла.

Осталось назвать расширения, под которыми сохраняются файлы того или иного формата. Файлы GIF, PNG и SVG имеют «говорящие» расширения gif, png и svg, а файлы JPEG — jpg, jpeg или jpe (встречается редко).

Вставка графических изображений

Поместить на страницу графическое изображение позволяет одинарный тег `<img>`. Web-обозреватель выведет изображение в том месте страницы, в котором находится этот тег.

Обязательный атрибут этого тега `src` служит для указания интернет-адреса файла с изображением:

```
<img src="image.png">
```

Рассматриваемый тег помещает на Web-страницу изображение, хранящееся в файле `image.gif`, который находится в той же папке, что и файл самой этой страницы.

Следующий тег помещает на страницу изображение из файла `picture.jpg`, который хранится в папке `images`, вложенной в корневую папку сайта:

```
<img src="/images/picture.jpg">
```

А этот тег помещает на страницу изображение, хранящееся в файле с интернет-адресом **http://www.othersite.ru/book12/author.jpg**, т. е. изображение с другого сайта:

```
<img src="http://www.othersite.ru/book12/author.jpg">
```

> ***На заметку***
>
> Принципы формирования интернет-адресов файлов, применяемые в WWW, мы подробно рассмотрим в *главе 5*.

Изображение, помещенное на страницу с помощью тега `<img>`, представляет собой встроенный элемент. Это значит, что мы можем вставить графическое изображение прямо в абзац:

```
<p>Эта картинка находится прямо в тексте: <img src="image.png"></p>
```

Если нам понадобится отобразить на странице отдельное, не связанное ни с каким абзацем, графическое изображение, мы можем поместить его в блочный элемент, например, в абзац:

```
<p><img src="/images/picture.jpg"></p>
```

Поскольку изображение хранится в отдельном от страницы файле, Web-обозревателю придется послать Web-серверу еще один запрос на его получение. Web-серверу нужно будет найти этот файл и отправить его Web-обозревателю. Файл должен загрузиться по сети. На все это требуется время. Если изображений на странице много, все они велики по размеру, а канал связи медленный, на открытие страницы понадобится значительное время. Может случиться так, что сама страница будет успешно загружена и отображена на экране, а изображения — еще нет. И Web-обозреватель вместо не загруженного еще изображения выведет на экран пустой прямоугольник.

Тег `<img>` поддерживает необязательный атрибут `alt`, с помощью которого указывается так называемый *текст замены*. Он будет выводиться в пустом прямоугольнике, обозначающем незагруженное изображение, пока это изображение не загрузится:

```
<p><img src="image.gif" alt="Картинка"></p>
```

> ***На заметку***
>
> Хорошим тоном Web-верстки считается указание текста замены только у значащих изображений. У изображений, являющихся элементами оформления Web-страницы, текст замены обычно не указывают.

На этом пока все об интернет-графике. Мы еще вернемся к ней в *главе 5*, когда станем рассматривать изображения-гиперссылки и карты-изображения. А сейчас пора начать разговор о...

Мультимедиа

Мультимедиа в приложении к WWW — это аудио и видео, размещенные на Web-страницах. Поскольку аудио- и видеоролики также хранятся в отдельных файлах, мультимедиа является внедренным элементом.

Поддерживаемые форматы интернет-мультимедиа

Форматов мультимедийных файлов существует не меньше, чем форматов файлов графических. Как и в случае с интернет-графикой, Web-обозреватели поддерживают далеко не все мультимедийные форматы, а только немногие.

Ранее, еще пару лет назад, поместить на страницу аудио- или видеоролик было большой проблемой. Дело в том, что все существовавшие тогда программы Web-обозревателей делились на две группы — по поддерживаемым ими форматам аудио и видео. Одна группа поддерживала аудиоформат MP3 и видеоформат MP4, а другая — OGG и WebM соответственно. Поэтому каждый ролик приходилось сохранять в двух разных форматах, чтобы посетители имели возможность прослушивать или просматривать его в разных Web-обозревателях.

Но, к счастью, эта «война форматов» уже в прошлом. Теперь, чтобы поместить на страницу аудио- или видеоролик, достаточно закодировать его в формате:

- MP3 — если это аудиоролик;
- MP4 — если это видеоролик.

Оба этих формата поддерживаются всеми имеющими сейчас хождение Web-обозревателями.

Вставка аудиоролика

Для вставки на страницу аудиоролика язык HTML 5 предлагает парный тег `<audio>`. Интернет-адрес файла, в котором хранится файл с роликом, указывают с помощью обязательного атрибута `src` этого тега:

```
<audio src="audio.mp3"></audio>
```

Содержимое самого тега будет выведено на экран только в том случае, если Web-обозреватель не поддерживает язык HTML 5. Этим можно воспользоваться, чтобы дать посетителю понять, что его программа для путешествий по Интернету устарела и должна быть обновлена.

Тег `<audio>` создает блочный элемент страницы.

По умолчанию Web-обозреватель не будет воспроизводить аудиоролик. Чтобы он это сделал, в теге `<audio>` нужно указать атрибут `autoplay`. Это особенный атрибут тега — он не имеет значения, достаточно одного его присутствия в теге, чтобы он начал действовать (*атрибут тега без значения*):

```
<p>Сейчас вы услышите звук!</p>
<audio src="audio.mp3" autoplay></audio>
```

По умолчанию аудиоролик никак не отображается на Web-странице (что, впрочем, понятно — аудио нужно не смотреть, а слушать). Но если в теге `<audio>` поставить атрибут без значения `controls`, Web-обозреватель выведет в том месте страницы, где проставлен этот тег, элементы для управления воспроизведением аудиоролика (рис. 3.1). Они включают кнопку запуска и приостановки воспроизведения, шкалу воспроизведения и регулятор громкости:

```
<p>Нажмите кнопку воспроизведения, чтобы услышать звук.</p>
<audio src="audio.mp3" controls></audio>
```

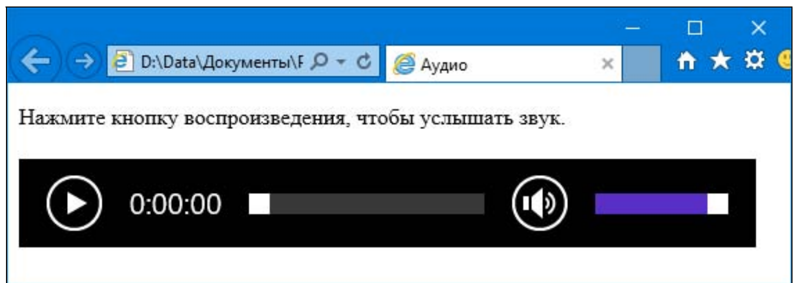


Рис. 3.1. Элементы для управления воспроизведением аудиоролика

Рассмотрим еще несколько полезных атрибутов тега `<audio>`.

- `loop` — атрибут без значения, указывает, что воспроизведение ролика должно продолжаться бесконечно («зацикленный» звук). В противном случае ролик будет воспроизведен всего один раз.

- muted — атрибут без значения, указывает, что звук должен быть изначально приглушен. Восстановить звук посетитель сможет, нажав соответствующую кнопку. Мы также можем восстановить звук программно, создав соответствующее поведение (о поведении страниц будет рассказано в *разделе 3*).
- preload — указывает Web-обозревателю выполнить предварительную загрузку ролика. Это позволяет устранить задержку между нажатием кнопки воспроизведения и собственно началом проигрывания. Поддерживаются три значения:
  - none — не выполнять предзагрузку;
  - metadata — загрузить только самое начало (заголовок) файла, где хранятся сведения о ролике, в частности, его продолжительность;
  - auto — загрузить файл целиком.

  Этот атрибут тега имеет смысл указывать лишь в том случае, если автоматическое воспроизведение не активизировано (атрибут autoplay отсутствует в теге).

Пример:

```
<audio src="audio.mp3" controls loop preload="auto"></audio>
```

Вставка видеоролика

Для вставки на Web-страницу видеоролика предназначен парный тег <video>. Интернет-адрес видеофайла указывается с помощью знакомого нам атрибута src этого тега:

```
<video src="video.mp4"></video>
```

Тег <video> также создает блочный элемент. Web-обозреватель, встретив его, выводит в этом месте страницы панель для просмотра видеоролика.

Тег <video> поддерживает те же атрибуты, что и тег <audio>: autoplay, controls, loop, muted и preload:

```
<video src="video.mp4" controls muted preload="metadata"></video>
```

И набор элементов для управления воспроизведением ролика аналогичен тому, который выводится тегом <audio>, — добавится лишь кнопка развертывания панели просмотра на все окно Web-обозревателя (рис. 3.2). Нужно только отметить, что через некоторое время после открытия страницы эти элементы управления исчезнут и появятся лишь при наведении на ролик курсора мыши.

Если воспроизведение видеоролика еще не запущено, в панели просмотра будет выведен первый его кадр или вообще ничего (конкретное поведение различается у разных Web-обозревателей). Но мы можем указать графическое изображение, которое будет туда выведено в качестве заставки. Для этого служит атрибут poster тега <video> — его значение указывает интернет-адрес нужного графического файла:

```
<video src="film.mp4" controls poster="filmposter.jpg"></video>
```

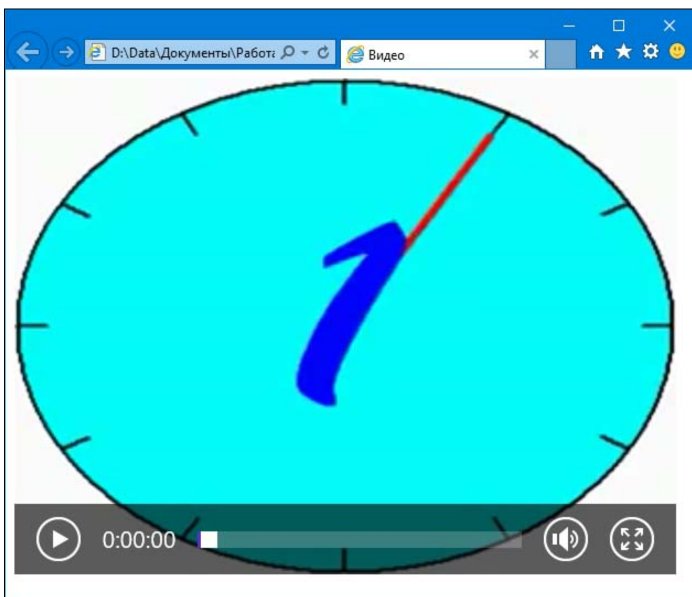
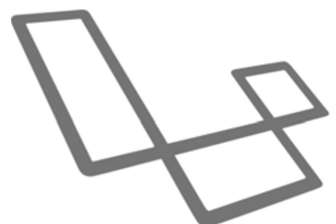


Рис. 3.2. Элементы для управления воспроизведением видеоролика

Здесь мы указали для видеоролика изображение-заставку, которое будет выведено в панели просмотра перед началом его воспроизведения и которое хранится в файле `filmposter.jpg`.



ГЛАВА 4

Таблицы

Что часто встречается в печатных изданиях, помимо текста и картинок? Таблицы! Таблицами пестрят учебники и справочники, таблицы нередко попадаются в газетах и журналах, даже художественные произведения иногда огорошивают нас таблицами. И это понятно. Таблицы — лучший способ уместить множество сведений на ограниченной площади страницы.

Предоставляет ли HTML средства для создания таблиц? Да, и уже довольно давно. Сейчас мы с ними познакомимся.

Создание таблиц

Таблица представляет собой блочный элемент. Это значит, что она размещается отдельно от всех остальных блочных элементов: абзацев, заголовков, больших цитат, аудио- и видеороликов. Так что вставить таблицу в абзац мы не сможем. (Нужно сказать, что таблица в абзаце выглядела бы, по меньшей мере, странно...)

Таблицы HTML создаются в четыре этапа.

На первом этапе в HTML-коде с помощью парного тега `<table>` формируют саму таблицу:

```
<table>
</table>
```

На втором этапе формируют строки таблицы. Для этого предусмотрены парные теги `<tr>` — каждый такой тег создает отдельную строку. Теги `<tr>` помещают внутрь тега `<table>` (листинг 4.1).

Листинг 4.1

```
<table>
  <tr>
  </tr>
  <tr>
  </tr>
```

```
  <tr>
  </tr>
</table>
```

На третьем этапе создают ячейки таблицы, для чего используют парные теги `<td>` и `<th>`. Тег `<td>` создает обычную ячейку, а тег `<th>` — *ячейку «шапки»*, в которой будет помещаться «шапка» соответствующего столбца таблицы. Теги `<td>` и `<th>` помещают в теги `<tr>`, создающие строки таблицы, в которых должны находиться эти ячейки (листинг 4.2).

Листинг 4.2

```
<table>
  <tr>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <tr>
    <td></td>
    <td></td>
    <td></td>
  </tr>
  <tr>
    <td></td>
    <td></td>
    <td></td>
  </tr>
</table>
```

На четвертом, последнем, этапе указывают содержимое ячеек, которое помещают в соответствующие теги `<td>` и `<th>` (листинг 4.3).

Листинг 4.3

```
<table>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
  </tr>
  <tr>
    <td>Ячейка 1.1</td>
    <td>Ячейка 1.2</td>
    <td>Ячейка 1.3</td>
  </tr>
```

```
  <tr>
    <td>Ячейка 2.1</td>
    <td>Ячейка 2.2</td>
    <td>Ячейка 2.3</td>
  </tr>
</table>
```

Приведенный в листинге 4.3 код создаст таблицу, которую можно увидеть на рис. 4.1.

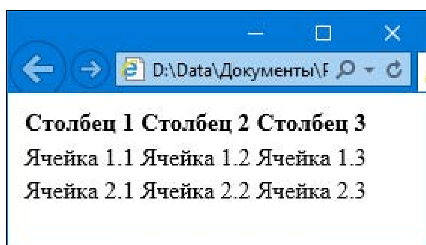


Рис. 4.1. Пример таблицы

Наша первая таблица не очень презентабельна. Но мы всегда можем задать какое-либо оформление для нее самой, для ее строк, столбцов, ячеек и содержимого, применив к ним соответствующее представление. Этим мы и займемся в *разделе 2*.

Если нам нужно поместить в ячейку таблицы простой текст, мы можем просто вставить его в соответствующий тег `<td>` или `<th>` (как показано в листинге 4.3). Если же нам потребуется нечто более сложное, мы можем использовать для структурирования и оформления ячеек таблицы теги, рассмотренные в *главе 2* (листинг 4.4).

Листинг 4.4

```
<table>
  . . .
  <tr>
    <td>
      <h4>Ячейка 1.1</h4>
      <p>В ячейку таблицы мы можем поместить что угодно.</p>
      <p>В том числе графику, аудио, видео и <em>даже другую
      таблицу</em>.</p>
    </td>
    . . .
  </tr>
  . . .
</table>
```

Здесь мы вставили в ячейку таблицы заголовок и два абзаца, причем фрагмент одного из этих абзацев пометили как важный.

Мы можем поместить в ячейку и графическое изображение:

```
<td><img src="picture.jpg" alt="Картинка в ячейке таблицы"></td>
```

А также, как говорилось ранее, аудио-, видеоролик и даже другую таблицу. В этом смысле HTML никак нас не ограничивает.

Теперь настала пора рассмотреть правила, которыми руководствуются Web-обозреватели при выводе таблиц на экран.

- Таблица представляет собой блочный элемент страницы (собственно, об этом мы уже говорили).
- Размеры таблицы и ее ячеек делаются такими, чтобы полностью вместить их содержимое.
- Между границами отдельных ячеек и между границей каждой ячейки и ее содержимым делается небольшой отступ.
- Текст ячеек заголовка выводится полужирным шрифтом и выравнивается по центру.
- Рамки вокруг всей таблицы и вокруг отдельных ее ячеек не рисуются.

И еще несколько правил, согласно которым создается HTML-код таблиц. Если их нарушить, Web-обозреватель отобразит таблицу некорректно или не отобразит ее вообще.

- Тег `<tr>` может находиться только внутри тега `<table>`. Любое другое содержимое тега `<table>` (кроме заголовка, секций таблицы, колонок и групп колонок, речь о которых пойдет далее) будет проигнорировано.
- Теги `<td>` и `<th>` могут находиться только внутри тега `<tr>`. Любое другое содержимое тега `<tr>` будет проигнорировано.
- Содержимое таблицы может находиться только в тегах `<td>` и `<th>`.
- Ячейки таблицы должны иметь хоть какое-то содержимое, иначе Web-обозреватель может их вообще не отобразить. Если же какая-то ячейка должна быть пустой, в нее следует поместить неразрывный пробел (HTML-литерал ` `).

Дополнительные инструменты для создания таблиц

А теперь рассмотрим дополнительные инструменты, которые можно использовать при создании таблиц. На практике они применяются нечасто, но в некоторых случаях без них не обойтись.

Заголовок таблицы

Прежде всего, с помощью парного тега `<caption>` мы можем дать таблице заголовок. Этот тег помещается внутрь тега `<table>`, в самом начале его содержимого.

В качестве содержимого тега <caption> указывается текст заголовка таблицы (листинг 4.5).

Листинг 4.5

```
<table>
  <caption>Это таблица</caption>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
  </tr>
  . . .
</table>
```

Заголовок таблицы выводится над самой таблицей и выравнивается по ее центру.

Секции таблицы

Кроме того, мы можем логически разбить таблицу HTML на три значащие части — *секции таблицы*:

- *секцию «шапки»*, в которой находится ее «шапка» (строки с ячейками «шапки»);
- *секцию тела*, где находятся строки таблицы, содержащие основные данные;
- *секцию «поддона»* с «поддоном» таблицы (строки с итоговыми данными, примечаниями и проч.).

Секцию «шапки» таблицы формирует тег <thead>, секцию тела — тег <tbody>, а секцию «поддона» — тег <tfoot>. Все эти теги парные, помещаются непосредственно в тег <table> и содержат теги <tr>, формирующие строки таблицы, которые входят в соответствующую секцию (листинг 4.6).

Листинг 4.6

```
<table>
  <thead>
    <tr>
      <th>Столбец 1</th>
      <th>Столбец 2</th>
      <th>Столбец 3</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Ячейка 1.1</td>
      <td>Ячейка 1.2</td>
      <td>Ячейка 1.3</td>
    </tr>
```

```
    <tr>
      <td>Ячейка 2.1</td>
      <td>Ячейка 2.2</td>
      <td>Ячейка 2.3</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>Итог по ячейке 2.1</td>
      <td>Итог по ячейке 2.2</td>
      <td>Итог по ячейке 2.3</td>
    </tr>
  </tfoot>
</table>
```

Секции таблицы Web-обозреватель никак не отображает и не выделяет на Web-странице. Они просто делят таблицу на три логические части. Однако мы можем задать для тегов, формирующих секции таблицы, какое-то представление. Кстати, секции таблицы, в основном, для этого и применяются.

Колонки и группы колонок

И, наконец, мы можем сформировать в таблице колонки и группы колонок. Это может пригодиться, если мы хотим задать представление для какого-либо отдельного столбца таблицы или сразу для нескольких ее столбцов.

Мы можем использовать в таблице либо только группы колонок, либо и колонки, и их группы. Это зависит от того, насколько сложное представление нам требуется задать. (Конкретные примеры использования колонок и их групп будут приведены в *главе 12*.)

Группы колонок использовать проще всего. Каждая такая группа может включать произвольное количество столбцов таблицы.

Группа колонок создается парным тегом `<colgroup>`. Содержимое у этого тега в таком случае не указывается. Набор тегов `<colgroup>`, описывающих группы колонок, помещается внутри тега `<table>`, в самом начале его содержимого. Если там присутствует тег `<caption>`, теги групп колонок записываются после него.

Тег `<colgroup>` поддерживает необязательный атрибут `span`, указывающий количество столбцов таблицы, что входят в состав группы. Если этот атрибут тега отсутствует, группа будет включать лишь один столбец.

В листинге 4.7 приведен фрагмент кода таблицы, включающей три группы колонок. Первая группа включает первый столбец, вторая — второй и третий, третья — четвертый.

Листинг 4.7

```
<table>
  <colgroup></colgroup>
  <colgroup span="2"></colgroup>
  <colgroup></colgroup>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
    <th>Столбец 4</th>
  </tr>
  . . .
</table>
```

Более сложный подход — применение и групп колонок, и колонок. В этом случае *колонка* включает либо столбец таблицы, либо сразу несколько столбцов, а группа колонок служит для объединения уже колонок.

Для создания колонки предусмотрен одинарный тег `<col>`. Необязательный атрибут этого тега `span` указывает количество входящих в колонку столбцов — если он отсутствует в теге, колонка включит лишь один столбец.

Теги колонок, которые следует объединить в одну группу, помещаются внутрь тега `<colgroup>`.

Листинг 4.8 показывает фрагмент кода таблицы, включающей три колонки, которые объединены в две группы. Первая колонка включает первый и второй столбцы, вторая — третий, третья — четвертый. Первая и вторая колонки включены в состав первой группы, третья входит во вторую группу колонок.

Листинг 4.8

```
<table>
  <colgroup>
    <col span="2">
    <col>
  </colgroup>
  <colgroup>
    <col>
  </colgroup>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
    <th>Столбец 4</th>
  </tr>
  . . .
</table>
```

Как и секции таблицы, колонки и группы колонок никак не выделяются визуально при выводе на экран. Они служат лишь для указания оформления ячеек таблицы, входящих в ее отдельные столбцы.

Объединение ячеек таблиц

Осталось поговорить еще об одной интересной возможности, предлагаемой языком HTML. Это так называемое *объединение ячеек* таблиц. Лучше всего рассмотреть пример — простую таблицу, HTML-код которой приведен в листинге 4.9.

Листинг 4.9

```
<table>
  <tr>
    <td>1</td>
    <td>2</td>
    <td>3</td>
    <td>4</td>
    <td>5</td>
  </tr>
  <tr>
    <td>6</td>
    <td>7</td>
    <td>8</td>
    <td>9</td>
    <td>10</td>
  </tr>
  <tr>
    <td>11</td>
    <td>12</td>
    <td>13</td>
    <td>14</td>
    <td>15</td>
  </tr>
  <tr>
    <td>16</td>
    <td>17</td>
    <td>18</td>
    <td>19</td>
    <td>20</td>
  </tr>
</table>
```

Это обычная таблица, ячейки которой пронумерованы, — так нам будет проще в дальнейшем. В окне Web-обозревателя она будет выглядеть так, как показано на рис. 4.2.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Рис. 4.2. Исходная таблица, чьи ячейки будут подвергнуты объединению

А теперь рассмотрим таблицу на рис. 4.3.

Здесь выполнено объединение некоторых ячеек, которые в результате слились в одну. Как это сделать?

<table>
<tr><td rowspan="2">1+6</td><td colspan="2">2+3</td><td colspan="2">4+5</td></tr>
<tr><td>7</td><td>8</td><td>9</td><td>10</td></tr>
<tr><td>11</td><td colspan="4">12+13+14+15</td></tr>
<tr><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td></tr>
</table>

Рис. 4.3. Таблица, показанная на рис. 4.2, после объединения некоторых ячеек (объединенные ячейки обозначены сложением их номеров)

Теги `<td>` и `<th>` поддерживают два весьма примечательных необязательных атрибута. Первый — `colspan` — объединяет ячейки по горизонтали, второй — `rowspan` — по вертикали.

Чтобы объединить несколько ячеек по горизонтали в одну, нужно выполнить следующие шаги.

1. Найти в коде HTML тег `<td>` (`<th>`), соответствующий первой из объединяемых ячеек (если считать ячейки в направлении слева направо).
2. Вписать в него атрибут `colspan` и указать в качестве его значения количество объединяемых ячеек, считая и самую первую из них.
3. Удалить теги `<td>` (`<th>`), создающие остальные объединяемые ячейки этой строки.

Давайте объединим ячейки 2 и 3 таблицы (см. листинг 4.9). Исправленный фрагмент кода, создающий первую строку этой таблицы, приведен в листинге 4.10.

Листинг 4.10

```
. . .
<tr>
  <td>1</td>
  <td colspan="2">2 + 3</td>
  <td>4</td>
  <td>5</td>
</tr>
. . .
```

Точно так же создадим объединенные ячейки 4 + 5 и 12 + 13 + 14 + 15.

Объединить ячейки по вертикали чуть сложнее. Вот шаги, которые нужно для этого выполнить.

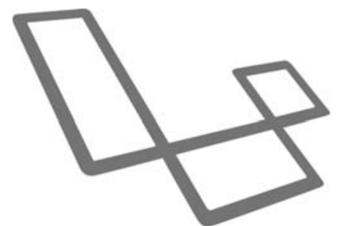
1. Найти в коде HTML строку (тег `<tr>`), в которой находится первая из объединяемых ячеек (если считать строки в направлении сверху вниз).
2. Найти в коде этой строки тег `<td>` (`<th>`), соответствующий первой из объединяемых ячеек.
3. Вписать в него атрибут `rowspan` и указать в качестве его значения количество объединяемых ячеек, считая и самую первую из них.
4. Просмотреть последующие строки и удалить из них теги `<td>` (`<th>`), создающие остальные объединяемые ячейки.

Нам осталось объединить ячейки 1 и 6 нашей таблицы. Листинг 4.11 показывает исправленный фрагмент ее HTML-кода (исправления затронут первую и вторую строки).

Листинг 4.11

```
. . .
<tr>
  <td rowspan="2">1 + 6</td>
  <td colspan="2">2 + 3</td>
  <td colspan="2">4 + 5</td>
</tr>
<tr>
  <td>7</td>
  <td>8</td>
  <td>9</td>
  <td>10</td>
</tr>
. . .
```

Обратим внимание, что мы удалили из второй строки тег `<td>`, создающий ячейку 6, поскольку она объединилась с ячейкой 1.



ГЛАВА 5

Средства навигации

В предыдущих главах мы наполняли Web-страницы содержимым: текстом, графическими изображениями, аудио-, видеороликами и таблицами. И, в принципе, изучили все возможности HTML по созданию содержания страниц.

Осталось связать эти разрозненные страницы воедино — в Web-сайт. Как это осуществить? С помощью средств навигации — гиперссылок.

Гиперссылка обычно выглядит как подчеркнутый фрагмент текста — если навести на него курсор мыши, он примет вид «указующего перста». При щелчке на гиперссылке Web-обозреватель загрузит страницу, интернет-адрес которой указан в параметрах этой гиперссылки (*целевую* Web-страницу). Гиперссылка также может иметь вид графического изображения или его фрагмента, такие гиперссылки сейчас довольно популярны.

Текстовые гиперссылки

Начнем с самых простых — *текстовых* гиперссылок, которые представляют собой фрагмент текста.

Создание гиперссылок

Создать текстовую гиперссылку очень просто. Достаточно найти в блочном элементе (например, абзаце) фрагмент текста, который нужно превратить в гиперссылку, и заключить его в парный тег `<a>`. Интернет-адрес целевой Web-страницы указывается в атрибуте `href` этого тега, который в таком случае является обязательным:

```
<a href="http://www.somesite.ru/pages/page125.html">Страница №125</a>
```

Эта гиперссылка указывает на страницу `page125.html`, хранящуюся в папке `pages` корневой папки в составе сайта `http://www.somesite.ru/`.

Следующая гиперссылка указывает на архивный файл `archive.zip`, хранящийся в той же папке, что и страница, которая в данный момент открыта в Web-обозревателе (*текущая* страница). При щелчке на гиперссылке Web-обозреватель

предложит загрузить этот архивный файл и либо открыть его, либо сохранить на диске клиентского компьютера:

```
<a href="archive.zip">Архив</a>
```

Гиперссылка (тег <a>) представляет собой встроенный элемент страницы, т. е. должна находиться в составе блочного элемента, например, абзаца:

```
<p><a href="22.html">Предыдущая страница</a>,
<a href="24.html">следующая страница</a>.</p>
```

Этот фрагмент HTML-кода создает абзац, содержащий сразу две гиперссылки, которые указывают на разные целевые страницы.

Текст, являющийся гиперссылкой, можно оформлять, используя любые теги, что описывались в *главе 2*:

```
<a href="http://www.somesite.ru/pages/page125.html">Страница
<em>№ 125</em></a>
```

Тег <a> поддерживает необязательный атрибут target. Он задает *цель гиперссылки*, указывающую, где будет открыта целевая страница. Так, если атрибуту тега target присвоить значение _blank, целевая страница будет открыта в новой вкладке или окне Web-обозревателя:

```
<a href="http://www.somesite.ru/pages/page125.html" target="_blank">
↳Страница № 125</a>
```

Чтобы задать обычное поведение гиперссылки (когда целевая страница открывается в той же вкладке или окне Web-обозревателя), нужно присвоить атрибуту тега target значение _self (это его значение по умолчанию) или вообще убрать этот атрибут из тега <a>.

Имеется также возможность создать гиперссылку, которая никуда не указывает (*«пустую»* гиперссылку). Для этого достаточно задать в качестве значения атрибута href значок # («решетка»):

```
<a href="#">А я никуда не веду!</a>
```

При щелчке на такой гиперссылке ничего не произойдет. «Пустые» гиперссылки часто применяются при создании интерактивных элементов страниц. Этим мы займемся в *разделе 3*.

Правила отображения гиперссылок Web-обозревателем:

- обычные гиперссылки выводятся синим цветом;
- гиперссылки, по которым посетитель уже «ходил» (*посещенные* гиперссылки), выводятся фиолетовым цветом;
- гиперссылка, по которой посетитель в данный момент щелкает (*активная* гиперссылка), выводится ярко-красным цветом;
- текст любых гиперссылок подчеркивается;
- при помещении курсора мыши на гиперссылку Web-обозреватель меняет его форму на «указующий перст».

Это внешний вид гиперссылок по умолчанию, который мы можем изменить, создав соответствующее представление. О том, как это сделать, будет рассказано в *разделе 2*.

Интернет-адреса в WWW

Об интернет-адресах файлов мы говорили еще в *главе 1*. Однако WWW привносит в них кое-что новое, что нам обязательно нужно знать.

Рассмотрим первый пример гиперссылки из предыдущего раздела. Ее интернет-адрес таков: **http://www.somesite.ru/pages/page125.html**. Он содержит и интернет-адрес сайта, и путь к файлу, который нужно получить, вследствие чего называется *полным*. Полные интернет-адреса используют, если нужно создать гиперссылку, указывающую на файл, входящий в состав другого сайта.

Однако, если гиперссылка указывает на файл того же сайта, к которому относится и файл текущей страницы, предпочтительнее *сокращенный* интернет-адрес, содержащий только путь к запрашиваемому файлу (интернет-адрес этого сайта и так известен Web-обозревателю).

Существуют два типа сокращенных интернет-адресов. Адреса первого типа задают путь к файлу, который нужно получить (*целевому* файлу), относительно корневой папки сайта. Они содержат в своем начале символ `/` (слеш), который и говорит Web-серверу, что путь нужно отсчитывать относительно корневой папки.

> ***На заметку***
>
> О корневой папке сайта было рассказано в *главе 1*. Вкратце: это особая папка, находящаяся на диске компьютера, на котором хранится сайт и работает Web-сервер, в этой папке должны помещаться все файлы сайта.

Например, интернет-адрес:

/page3.html

указывает на файл **page3.html**, хранящийся непосредственно в корневой папке сайта.

А интернет-адрес:

/articles/article1.html

указывает на файл **article1.html**, хранящийся в папке **articles**, вложенной в корневую папку сайта.

Такие интернет-адреса называют *абсолютными* и используют, если нужно создать гиперссылку на файл, хранящийся в «глубине» сайта (скажем, в другой папке, нежели файл текущей страницы).

Сокращенные интернет-адреса второго типа задают путь к целевому файлу относительно файла текущей страницы. Они не содержат в начале символа слеша — и в этом их важное отличие от абсолютных интернет-адресов.

Рассмотрим несколько примеров подобных интернет-адресов.

- **archive.zip**

  Этот интернет-адрес указывает на файл **archive.zip**, хранящийся в той же папке, что и файл текущей страницы.

- **chapter3/page1.html**

  Этот интернет-адрес указывает на страницу **page1.html**, хранящуюся в папке **chapter3**, что вложена в папку, в которой хранится текущая страница.

- **../contents.html**

  А этот интернет-адрес указывает на страницу **contents.html**, хранящуюся в папке, в которую вложена папка, где хранится текущая страница. (Обратим внимание на две точки в начале пути — так обозначается переход на папку предыдущего уровня вложенности.)

Такие интернет-адреса называют *относительными*. Их применяют, если нужно создать гиперссылку на файл, хранящийся в той же папке, что и текущая страница, одной из вложенных в нее папок или папке предыдущего уровня вложенности.

> ***На заметку***
>
> Во многих случаях лучше поэкспериментировать с разными интернет-адресами, чтобы выяснить, какой именно подойдет — абсолютный или относительный.

> ***Внимание!***
>
> В страницах, которые не должны быть опубликованы на Web-серверах, а будут открываться с диска клиентских компьютеров, следует применять только относительные интернет-адреса. Дело в том, что файловая система компьютера не знает, какую папку считать корневой, поэтому не сможет правильно интерпретировать абсолютные интернет-адреса. (Разумеется, гиперссылки, ссылающиеся на другие сайты, должны содержать полные интернет-адреса.)

Почтовые гиперссылки

HTML позволяет нам создать гиперссылку, указывающую на адрес электронной почты (*почтовую гиперссылку*). Щелчок на ней запустит программу почтового клиента, помеченную в настройках системы в качестве используемой по умолчанию.

Интернет-адрес такой гиперссылки записывается в следующем формате:

mailto:<*адрес электронной почты*>

причем между двоеточием после **mailto** и собственно адресом не должно быть пробелов:

```
<a href="mailto:user@mailserver.ru">Отправить письмо</a>
```

Дополнительные возможности гиперссылок

Язык HTML предлагает нам некоторые дополнительные возможности для создания гиперссылок. Их применяют нечасто, но иногда они бывают полезны.

Прежде всего, мы можем указать для гиперссылки *«горячую» клавишу*. Если посетитель нажмет эту клавишу, удерживая нажатой клавишу <Alt>, Web-обозреватель выполнит переход по такой гиперссылке.

Для указания «горячей» клавиши предусмотрен необязательный атрибут `accesskey` тега `<a>`. Значение этого атрибута — латинская буква, соответствующая нужной клавише:

```
<a href="http://www.somesite.ru/pages/page125.html"
accesskey="d">Страница № 125</a>
```

Здесь мы указали для гиперссылки «горячую» клавишу <D>. И, чтобы перейти по ней, посетителю будет достаточно нажать комбинацию клавиш <Alt>+<D>.

На гиперссылках можно щелкать мышью — так поступает большинство пользователей. Но по ним также можно «путешествовать» с помощью клавиатуры. В таком случае говорят о *фокусе ввода*, получаемом гиперссылкой, — признаке того, что эта гиперссылка будет реагировать на нажатия клавиш. Гиперссылка, имеющая фокус ввода, выделяется тонкой черной штриховой рамкой.

- Если нажать клавишу <Enter>, Web-обозреватель выполнит переход по гиперссылке, имеющей в данный момент фокус ввода.
- Если нажать клавишу <Tab>, Web-обозреватель перенесет фокус ввода на следующую гиперссылку.
- Если нажать комбинацию клавиш <Shift>+<Tab>, Web-обозреватель перенесет фокус ввода на предыдущую гиперссылку.

Порядок, в котором выполняется перенос фокуса ввода с одной гиперссылки на другую при нажатии клавиш <Tab> или <Shift>+<Tab>, носит название *порядка обхода*. По умолчанию он совпадает с порядком, в котором гиперссылки определены в HTML-коде страницы. Но мы можем указать свой порядок обхода с помощью необязательного атрибута `tabindex` тега `<a>`. Его значение — целое число от –32 767 до 32 767 — *номер в порядке обхода*.

- Если указан положительный номер, именно он будет определять порядок обхода. Иными словами, сначала фокус ввода получит гиперссылка с номером 1, потом — с номером 2, затем — с номером 3 и т. д.
- Если указан номер, равный нулю, обход будет осуществляться в порядке, в котором гиперссылка определена в HTML-коде страницы. Фактически ноль — значение атрибута тега `tabindex` по умолчанию.
- Если указан отрицательный номер, такая гиперссылка вообще исключается из порядка обхода. До нее невозможно будет добраться с помощью клавиатуры — можно будет только щелкать мышью.

Рассмотрим небольшой пример:

```
<a href="page1.htm" tabindex="3">Страница 1</a>
<a href="page2.htm" tabindex="2">Страница 2</a>
<a href="page3.htm" tabindex="1">Страница 3</a>
```

Этот HTML-код создает три гиперссылки с «обратным» порядком обхода. Сначала фокус ввода получит гиперссылка `Страница 3`, потом — `Страница 2` и напоследок — `Страница 1`.

Графические гиперссылки

В начале этой главы говорилось, что гиперссылка может быть сделана на основе не только фрагмента текста, но и картинки, и даже может представлять собой фрагмент графического изображения. Вот *графическими гиперссылками* мы сейчас и займемся.

Изображения-гиперссылки

Язык HTML позволяет создать гиперссылку на основе графического изображения, создав тем самым *изображение-гиперссылку*:

```
<a href="http://www.w3.org"><img src="w3logo.gif"></a>
```

Этот HTML-код создает изображение-гиперссылку, указывающую на Web-сайт организации *W3C* (WWWC, World Wide Web Consortium, Консорциум Всемирной паутины), которая занимается разработкой и поддержкой интернет-технологий, в том числе и языка HTML. А в качестве самого изображения выбран логотип этой организации, который мы сохранили в файле в той же папке, где находится файл текущей страницы.

А следующий HTML-код создает почтовую изображение-гиперссылку:

```
<a href="mailto:user@mailserver.ru"><img src="email.gif"></a>
```

Современные Web-обозреватели никак не выделяют изображение-гиперссылку при выводе на экран. Однако курсор мыши при наведении на нее все так же принимает вид «указующего перста».

Изображения-карты

А еще HTML позволяет превратить в гиперссылку часть графического изображения. Более того, мы можем разбить изображение на части, каждая из которых будет представлять собой отдельную гиперссылку, указывающую на свой интернет-адрес. Такие изображения называют *изображениями-картами*, а ее части-гиперссылки — *«горячими» областями*.

С помощью изображений-карт часто создают заголовки Web-сайтов, фрагменты которых представляют собой гиперссылки, указывающие на определенную страницу. Пик популярности изображений-карт давно прошел, однако их еще можно довольно часто встретить.

Изображение-карту создают в три этапа. Первый этап — создание самого изображения с помощью все того же тега `<img>`:

```
<img src="map.png">
```

Второй этап — создание *карты*, особого элемента страницы, который описывает набор «горячих» областей изображения-карты. Сама карта на странице никак не отображается.

Карту создают с помощью невидимого парного тега <map>:

```
<map name="<имя карты>">
</map>
```

С помощью обязательного атрибута name тега <map> задается уникальное в пределах страницы имя карты. Оно однозначно идентифицирует эту карту, может содержать только латинские буквы, цифры и знаки подчеркивания и начинаться должно с буквы:

```
<map name="samplemap">
</map>
```

После создания карты следует привязать ее к созданному на первом этапе изображению. Для этого мы применим обязательный в таком случае атрибут usemap тега <img>. Его значение — имя привязываемой к изображению карты, причем в начале этого имени обязательно следует поставить символ # («решетка»). (В имени, заданном атрибутом name тега <map>, символ # присутствовать не должен.)

```
<img src="map.png" usemap="#samplemap">
```

На третьем этапе создают описания самих «горячих» областей в карте. Их помещают внутрь описывающего карту тега <map> и формируют с помощью одинарных тегов <area>, которые записываются в формате:

```
<area [shape="rect|circle|poly"] coords="<набор параметров>"
↳href="<интернет-адрес гиперссылки>"|nohref
↳target="<цель гиперссылки>">
```

Необязательный атрибут этого тега shape задает форму «горячей» области, а обязательный атрибут coords — ее местоположение и размеры. Атрибут тега shape может принимать следующие значения:

- rect — прямоугольная «горячая» область. Значение атрибута тега coords в этом случае записывается в формате *<X1>*, *<Y1>*, *<X2>*, *<Y2>*, где X1 и Y1 — координаты верхнего левого, а X2 и Y2 — правого нижнего угла прямоугольника. Кстати, если атрибут тега shape отсутствует, создается именно прямоугольная область;
- circle — круглая «горячая» область. В этом случае значение атрибута тега coords записывается в формате *<X центра>*, *<Y центра>*, *<радиус>*;
- poly — «горячая» область в виде многоугольника. Атрибут тега coords должен иметь значение вида *<X1>*, *<Y1>*, *<X2>*, *<Y3>*, *<X3>*, *<Y3>*. . ., где X*n* и Y*n* — координаты соответствующей вершины многоугольника.

Атрибут href тега <area> задает интернет-адрес целевого файла. Он может быть заменен атрибутом без значения nohref, задающим область, не связанную ни с каким интернет-адресом. Это позволяет создавать оригинальные изображения-карты, например, карту в виде бублика, «дырка» которого никуда не указывает.

Также знакомый нам атрибут тега target задает цель гиперссылки. (Конечно, указывать его имеет смысл только в том случае, если мы создаем именно «горячую» область, а не «дырку» с атрибутом nohref.)

Листинг 5.1 содержит полный HTML-код, создающий изображение-карту.

Листинг 5.1

```
<img src="map.png" usemap="#samplemap">
. . .
<map name="samplemap">
  <area shape="circle" coords="50,50,30" href="page1.html">
  <area shape="circle" coords="50,150,30" href="page2.html">
  <area shape="poly" coords="100,50,100,100,150,50,100,50" nohref>
  <area shape="rect" coords="0,100,30,100" href="appendix.html"
  target="_blank">
</map>
```

Здесь мы создали две круглые «горячие» области, указывающие на страницы `page1.html` и `page2.html`, многоугольную область, не ссылающуюся никуда, и прямоугольную область, ссылающуюся на страницу `appendix.html`. Причем последняя «горячая» область при щелчке на ней откроет страницу в новом окне Web-обозревателя.

Панель навигации

Гиперссылки не всегда «ходят поодиночке». Довольно часто на Web-страницах присутствуют целые наборы гиперссылок, ведущих на разные страницы сайта. Такие наборы называются *панелями навигации*.

Панель навигации может быть горизонтальной, располагаться вверху или внизу страницы, или вертикальной, находящейся слева или справа. Вертикальная панель навигации обычно формируется с помощью набора блочных контейнеров (каждая гиперссылка помещается в отдельный блочный контейнер), в виде списка (гиперссылки представляют собой пункты этого списка) или таблицы (гиперссылки помещаются в ее ячейки). Горизонтальная панель навигации (или *полоса навигации*) может быть сформирована с помощью обычного абзаца, блока, таблицы или списка, для которого в этом случае придется задать соответствующее представление.

Гиперссылки панели навигации могут быть текстовыми или графическими. В последнем случае практически всегда применяют изображения-гиперссылки. Зачастую для изображений-гиперссылок реализуют особое поведение, заменяющее изображение другим при наведении на соответствующую гиперссылку курсора мыши. (О поведении страницы мы поговорим в *разделе 3*.)

Панель навигации всегда оформляют особым образом, чтобы привлечь к ней внимание посетителя. Ее можно выделить цветом текста и фона, рамкой, увеличенным размером шрифта или всем вместе. Все это реализуется с помощью задания соответствующего представления.

Якоря

Напоследок рассмотрим еще одну возможность, предлагаемую нам языком HTML и способную сильно упростить посетителям чтение длинных текстов. Хотя она и не относится к гиперссылкам напрямую, но действует совместно с ними.

Это так называемые *якоря* (anchors), о которых уже упоминалось в *главе 1*. Такой якорь помечает некоторый фрагмент Web-страницы, чтобы другая гиперссылка могла на него сослаться. Так, можно пометить отдельные главы длинного текстового документа, и посетитель сможет «перескочить» к нужной ему главе, щелкнув гиперссылку в оглавлении. Очень удобно!

Создать якорь очень просто. Достаточно указать для элемента страницы уникальное имя, задав его в атрибуте тега `id`, — и этот элемент станет якорем. К имени элемента страницы предъявляются те же требования, что и к имени карты (см. *разд. «Изображения-карты»* ранее в этой главе).

Листинг 5.2 иллюстрирует пример HTML-кода, создающего якорь.

Листинг 5.2

```
. . .
<p>Окончание второй главы...</p>
<h2 id="chapter3">Глава 3<h2>
<p>Начало третьей главы...</p>
. . .
```

Здесь мы превратили в якорь с именем `chapter3` заголовок второго уровня, предваряющий третью главу текста.

Хорошо! Якорь готов. Как теперь на него сослаться с другой страницы? Очень просто. Для этого достаточно создать обычную гиперссылку, добавив в ее интернет-адрес имя нужного нам якоря. Имя якоря ставят в самый конец интернет-адреса и отделяют от него символом `#` («решетка») — собственно, об этом говорилось еще в *главе 1*.

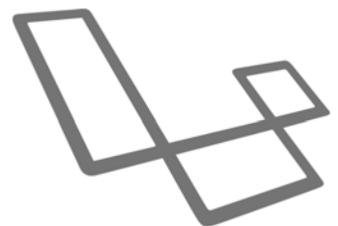
Предположим, что страница, содержащая якорь `chapter3`, хранится в файле `novel.html`. Тогда, чтобы сослаться на этот якорь с другой Web-страницы, мы создадим на последней такую гиперссылку:

```
<a href="novel.html#chapter3">Глава 3</a>
```

При щелчке на такой гиперссылке Web-обозреватель откроет страницу `novel.html` и прокрутит ее в окне так, чтобы достичь места, где находится якорь `chapter3`.

Если же нам нужно сослаться на якорь с той же страницы, где он находится, то можно использовать в качестве интернет-адреса только имя этого якоря, предварив его символом «решетки»:

```
<a href="#chapter3">Глава 3</a>
```



ГЛАВА 6

Web-формы и элементы управления. Фреймы

Поскольку мы собираемся разрабатывать динамический сайт, нам потребуется каким-то образом получать от посетителя информацию, предназначенную для занесения в базу данных. Понятно, что для этого следует разместить на странице всевозможные *элементы управления*: поля ввода, области редактирования, флажки, переключатели, списки и кнопки.

В этой главе мы научимся эти элементы управления создавать средствами HTML. А еще мы познакомимся с Web-формами, фреймами и изучим еще несколько терминов, без которых не поймем материал, изложенный в *части II*.

Web-формы

И начнем мы с изучения Web-форм как более фундаментальной составной части механизмов HTML, предназначенных для отправки серверной программе введенных посетителем данных.

Что такое Web-форма и зачем она нужна?

Web-форма, или просто *форма*, — это особый элемент страницы, служащий «вместилищем» для элементов управления. Web-форма выполняет сбор данных, введенных в элементы управления, их кодирование согласно указанному нами методу и отправку серверной программе, также согласно методу, который мы зададим.

Сразу же отметим, что каждый элемент управления должен иметь уникальное в пределах Web-формы имя. Это нужно для успешного формирования пар данных перед отправкой их серверной программе.

Начнем с *методов отправки данных*, поддерживаемых формами. Всего их два.

- *GET* — формирует из введенных посетителем данных набор пар вида:

  ```
  <имя элемента управления>=<введенное в него значение>
  ```

  Эти пары добавляются справа к интернет-адресу серверной программы и отделяются от последнего символом `?` (вопросительный знак), сами же пары разделяются символами `&` (амперсанд).

Если введенные посетителем данные содержат символы, недопустимые в интернет-адресах (например, пробелы, вопросительные знаки, амперсанды или символы кириллицы), они кодируются особым образом. В частности, пробелы заменяются знаками +, а прочие символы — их кодами.

Полученный таким образом интернет-адрес отправляется Web-серверу, который извлекает из него путь к файлу серверной программы и сами данные, после чего запускает программу и передает ей полученные данные.

Это те самые *GET-параметры*, которые упоминались в *главе 1*.

Метод GET обычно применяется для отправки данных небольшого объема: ключевых слов, по которым будет выполняться поиск, номеров страниц и т. п.

- *POST* — предварительно кодирует данные согласно указанному способу, после чего формирует из них аналогичные пары:

  ```
  <имя элемента управления>=<введенное в него значение>
  ```

  но не добавляет их к интернет-адресу серверной программы, а отправляет отдельно от него, хоть и в составе того же самого запроса.

  Методом POST отправляются данные значительного объема и сведения, которые следует сохранить в тайне: имена и пароли пользователей, регистрационные данные, тексты статей, публикуемых в Интернете, и др.

Если данные отправляются серверной программе с применением метода POST, они должны быть закодированы с применением одного из трех *методов кодирования данных*:

- `application/x-www-form-urlencoded` — все пробелы заменяются знаками +, все недопустимые в интернет-адресах символы заменяются их кодами (как видим, данные кодируются таким же образом, как и при отправке методом GET). Этот метод используется в большинстве случаев;
- `multipart/form-data` — данные фактически отправляются в том виде, в каком их ввел посетитель. Этот метод используется только при отправке через Web-форму серверной программе файлов (поскольку содержимое файлов не следует изменять при отправке);
- `text/plain` — пробелы заменяются знаками + и более никаких перекодировок не выполняется. Метод применяется крайне редко и в специальных случаях.

Разумеется, интернет-адрес серверной программы, которая должна получить занесенные посетителем данные, также указывается в форме. Как именно, мы скоро узнаем.

Создание Web-форм

Web-форма создается парным тегом `<form>`. Внутри него помещают теги, формирующие элементы управления, которые входят в эту форму:

```
<form>
  <теги, формирующие элементы управления>
</form>
```

Различные параметры формы указываются в атрибутах тега `<form>` (все эти атрибуты являются необязательными):

- `action` — указывает интернет-адрес серверной программы, которой будут отправлены данные.

  Если атрибут не указан, данные будут отправлены той же программе, что сгенерировала страницу, содержащую эту форму. Если же форма находится в составе статической страницы, последняя будет перезагружена;
- `method` — задает метод отправки данных, который указывается в виде одного из «говорящих» значений: `get` или `post`. Если не указан, используется метод отправки данных GET;
- `enctype` — задает метод кодирования данных. Имеет смысл лишь в случае отправки данных методом POST. Если не указан, используется метод кодирования `application/x-www-form-urlencoded`;
- `target` — знакомый нам по *главе 5* атрибут тега, указывающий, куда будет выведена страница, сгенерированная серверной программой после обработки переданных ей формой данных (обычно это страница с результатами их обработки). Если не указан, данные будут выведены в текущую вкладку или в окно Web-обозревателя;
- `name` — задает имя формы. Используется лишь в особых случаях;
- `autocomplete` — включает или отключает функцию автозаполнения для формы. Значение `on` включает автозаполнение, значение `off` отключает его. Значение по умолчанию — `on`;
- `novalidate` — атрибут тега без значения, отключающий проверку введенных в элементы управления данных на корректность.

Рассмотрим три примера кода, создающего Web-формы.

- Эта форма отправит данные серверной программе `register.php`, хранящейся в папке `apps` корневой папки сайта, с применением метода отправки POST и метода кодирования `application/x-www-form-urlencoded` (поскольку таковой не указан явно):

  ```
  <form action="/apps/register.php" method="post">
    . . .
  </form>
  ```
- Эта форма отправит данные серверной программе `upload.php`, но уже с применением метода кодирования `multipart/form-data`. Также мы отключили у формы автозаполнение:

  ```
  <form action="/apps/upload.php" method="post"
  enctype="multipart/form-data" autocomplete="off">
    . . .
  </form>
  ```

- А эта форма отправит данные действию `create` контроллера `articles` с применением метода GET, поскольку метод отправки не указан явно (о контроллерах и действиях разговор пойдет в *главе 29*):

```
<form action="/articles/create">
  . . .
</form>
```

Web-форма представляет собой блочный элемент страницы. На странице она никак не отображается (если, конечно, мы не зададим для нее какого-либо представления). В этом форма схожа с блочным контейнером, рассмотренным нами в *главе 2*.

Элементы управления

А теперь рассмотрим создание собственно элементов управления, поддерживаемых HTML.

Общие вопросы создания элементов управления

Большинство элементов управления HTML создаются посредством одинарного тега `<input>`. Некоторые элементы управления — такие как область редактирования и список — формируются другими тегами.

В теге `<input>` тип создаваемого элемента управления указывается в необязательном атрибуте `type`. Если этот атрибут тега отсутствует, или заданный в нем тип не «знаком» Web-обозревателю, будет создано обычное поле ввода.

Все теги, создающие элементы управления, поддерживают ряд других атрибутов, устанавливающих различные параметры создаваемых элементов. Давайте их рассмотрим:

- `name` — задает имя элемента управления, которое будет использовано для формирования пар:

  <имя элемента управления>=<введенное в него значение>

  В данном случае этот атрибут тега является обязательным;
- `required` — атрибут тега без значения, указывающий, что в этот элемент обязательно должно быть введено какое-либо значение. В противном случае форма не выполнит отправку данных;
- `readonly` — атрибут тега без значения, делающий элемент управления доступным только для чтения;
- `disabled` — атрибут тега без значения, делающий элемент полностью недоступным;
- `autofocus` — атрибут тега без значения, указывающий, что элемент управления сразу после загрузки страницы должен получить фокус ввода (о фокусе ввода говорилось в *главе 6*).

> **ВНИМАНИЕ!**
>
> Атрибут тега `autofocus` следует указывать только для одного элемента управления на всей странице.

Еще такие теги поддерживают атрибуты тегов `accesskey` и `tabindex`, знакомые нам по *главе 6*.

Все элементы управления HTML представляют собой встроенные элементы страницы. Поэтому, чтобы вывести их построчно, один за другим, применяют какие-либо блочные элементы, — обычно абзацы или блоки.

Поле ввода

Поле ввода в Web-формах применяется чаще всего. Создается оно с помощью одинарного тега `<input>`, атрибут `type` которого либо не указан, либо имеет значение `text`:

```
<input name="username" required>
```

Этот код создаст поле ввода `username`, обязательное для заполнения.

Для указания параметров, характерных для поля ввода, мы используем следующие необязательные атрибуты тега:

- `value` — задает значение, которое должно присутствовать в поле ввода изначально. Если атрибут не указан, поле ввода не будет содержать ничего;
- `maxlength` — задает максимальную длину значения, которое может быть занесено в поле ввода, в символах. Если не указан, в поле ввода можно будет занести значение неограниченной длины;
- `pattern` — указывает шаблон для проверки значения, заносимого в поле. При отправке данных форма сравнит введенное значение с шаблоном и выполнит отправку данных только в том случае, если значение совпадает с ним. Шаблон указывается в виде регулярного выражения (см. *главу 21*);
- `placeholder` — задает подсказку для посетителя, которая будет выведена прямо в поле ввода. Обычно такая подсказка указывает посетителю род и формат заносимых в поле данных;
- `autocomplete` — аналогичен одноименному атрибуту тега `<form>`.

Рассмотрим пример формы с двумя полями ввода (листинг 6.1).

Листинг 6.1

```
<form action="/apps/register.php" method="post">
  <p>Имя: <input name="name1" maxlength="20" autofocus></p>
  <p>Фамилия: <input name="name2" maxlength="30"></p>
</form>
```

Поскольку любые элементы управления — суть встроенные элементы, для вывода полей ввода построчно здесь используются абзацы.

Поле ввода пароля

Поле ввода пароля ничем не отличается от обычного поля ввода за тем исключением, что вместо вводимых символов в нем отображаются точки. Такие поля ввода широко применяют для указания паролей и других конфиденциальных данных.

Поле ввода пароля также создается с помощью одинарного тега `<input>`. Значением атрибута `type` этого тега в таком случае должно быть `password`.

Чтобы указать параметры поля ввода пароля, мы можем использовать те же атрибуты тега, что и для указания параметров обычного поля ввода.

В листинге 6.2 представлен код, создающий форму для входа на сайт, которая содержит обычное поле ввода (для указания имени пользователя) и поле ввода пароля (для задания пароля).

Листинг 6.2

```
<form action="/apps/login.php" method="post">
  <p>Имя: <input type="text" name="login" maxlength="20"></p>
  <p>Пароль: <input type="password" name="password"
  maxlength="20"></p>
</form>
```

Поле ввода числа

Поле ввода числа, как следует из его названия, позволяет вводить лишь числа. Если занести в это поле что-либо, не являющееся числом, форма не выполнит отправку данных.

Чтобы создать такое поле ввода, нужно указать для атрибута `type` тега `<input>` значение `number`. Для задания изначального значения мы можем использовать атрибут тега `value`. Также поле ввода числа поддерживает следующие необязательные атрибуты тега:

- `min` — задает минимальное число, которое может быть введено в поле;
- `max` — задает максимальное число, которое может быть введено в поле;
- `step` — задает интервал между допустимыми значениями.

Чтобы лучше понять назначение всех этих атрибутов тега, давайте рассмотрим такой пример:

```
<input type="number" name="cost" min="100" max="200" step="10">
```

В поле ввода, созданном этим тегом, можно будет занести числа 100, 110, 120 и т. д. до 200.

Поле ввода интернет-адреса

Поле ввода интернет-адреса перед отправкой данных будет проверять, занесен ли в него правильно составленный интернет-адрес. Если это не так, отправка данных формой будет блокирована.

Чтобы создать поле ввода интернет-адреса, у атрибута `type` тега `<input>` следует указать значение `url`. Для задания дополнительных параметров можно использовать атрибуты тега, знакомые нам по обычному полю ввода:

```
<input type="url" name="site_url" placeholder="Интернет-адрес сайта">
```

Поле ввода адреса электронной почты

Поле ввода адреса электронной почты перед отправкой проверяет, занесен ли в него корректно составленный адрес электронной почты.

Чтобы создать такое поле, необходимо дать атрибуту `type` тега `<input>` значение `email`. В остальном это поле аналогично обычному полю ввода:

```
<input type="email" name="user_mail" maxlength="127">
```

Флажок

Флажок создается так же, как и поле ввода, — тегом `<input>`. Вот только значением атрибута тега `type` должно быть `checkbox`:

```
<p><input type="checkbox" name="iagree"> Я согласен с условиями</p>
```

Атрибут тега без значения `checked` делает флажок изначально установленным (по умолчанию он сброшен).

Атрибут тега `value` задает значение, которое отправится серверной программе в случае, если флажок установлен. Допустим, если мы укажем для флажка значение `ok`:

```
<p><input type="checkbox" name="iagree" value="ok">
Я согласен с условиями</p>
```

то серверная программа получит пару `iagree=ok`.

Если атрибут `value` отсутствует в теге `<input>`, серверной программе будет отправлено значение `on`. А если флажок сброшен, серверная программа вообще не получит от него никаких данных.

Ранее говорилось, что каждый элемент управления в форме должен иметь уникальное имя. Однако HTML допускает создавать несколько флажков, имеющих одно и то же имя (но разные значения):

```
<input type="checkbox" name="mode" value="1">
<input type="checkbox" name="mode" value="2">
```

В таком случае серверная программа успешно получит данные сразу от всех установленных флажков. Так, если оба флажка, чей HTML-код здесь показан, установлены, то серверная программа получит пары `mode=1` и `mode=2`.

Переключатель

Переключатель создается тем же тегом `<input>`, только в его случае атрибуту `type` следует дать значение `radio`.

Переключатели всегда применяются группами, поскольку от одного-единственного переключателя нет никакой пользы (более того, он обескуражит посетителя). Переключатели, входящие в состав одной группы, должны иметь одно и то же имя, но при этом для них должны быть заданы разные значения.

Значение переключателя задается атрибутом тега `value`. Атрибут без значения `checked` позволяет сделать переключатель изначально установленным — таковым следует делать один переключатель во всей группе.

В листинге 6.3 приведен пример HTML-кода, создающий два переключателя. Они объединены в группу — об этом говорит одинаковое значение `gender`, заданное для атрибутов `name` их тегов. Для переключателей заданы значения `m` и `f`, а первый переключатель сделан изначально установленным.

Листинг 6.3

```
<p>Пол: <input type="radio" name="gender" value="m" checked> муж.
<input type="radio" name="gender" value="f"> жен.</p>
```

Серверная программа получит данные только от того переключателя, который на момент отправки был установлен. Так, если мы установим второй переключатель из созданных нами, серверная программа получит пару `gender=f`. Если же ни один переключатель группы не установлен (хотя так делать не рекомендуется), серверная программа вообще не получит данные от переключателей.

Регулятор

Регулятор HTML (рис. 6.1) аналогичен одноименному элементу управления в Windows-приложениях. Он представляет собой шкалу, по которой перемещается указатель, применяется для задания числовых значений и во многих случаях может быть использован вместо поля ввода числа.

Рис. 6.1. Регулятор

Создать регулятор можно, задав для атрибута `type` тега `<input>` значение `range`. Для установки параметров регулятора можно использовать те же атрибуты тега, что применяются в поле ввода числа: `min`, `max` и `step`:

```
<input type="range" name="mark" min="1" max="10" value="5">
```

Этот код создает регулятор, позволяющий выбирать числа от 1 до 10 и изначально имеющий значение 5.

Область редактирования

Область редактирования, в отличие от рассмотренных нами ранее элементов управления, создается парным тегом `<textarea>`. Изначальное значение, которое должно присутствовать в области редактирования и которое должно представлять собой обычный текст, помещается внутрь этого тега:

```
<textarea name="content">Введите текст статьи</textarea>
```

Поддерживается атрибут тега `placeholder`:

```
<textarea name="content" placeholder="Введите текст статьи"></textarea>
```

Список

Списки, как обычные, так и раскрывающиеся, создаются парным тегом `<select>`. Внутри этого тега помещают HTML-код, формирующий пункты списка (об их создании поговорим чуть позже):

```
<select>
  <код, создающий пункты списка>
</select>
```

Тег `<select>` поддерживает два необязательных атрибута, которыми устанавливаются два важных параметра:

- `size` — задает высоту списка в пунктах (имеются в виду пункты списка). Если он имеет значение, отличное от 1, создается обычный список, имеющий высоту, равную указанному значению пунктов. Если же этот атрибут тега имеет значение 1, создается раскрывающийся список (имеющий в высоту один пункт). Значение по умолчанию — 1, если в списке можно выбрать лишь один пункт, и 4, если в нем можно выбрать сразу несколько пунктов.
- `multiple` — атрибут без значения, разрешающий посетителю выбирать в этом списке сразу несколько пунктов.

Отдельный пункт списка формируется парным тегом `<option>`. Этот тег допускается указывать только в теге `<select>` (а также в теге `<optgroup>`, с которым мы познакомимся очень скоро).

Сам текст пункта помещается прямо в тег `<option>`. Другие параметры пункта задаются соответствующими атрибутами этого тега, которые нам обязательно нужно знать (все эти атрибуты тега являются необязательными):

- `value` — задает значение, которое будет отправлено серверной программе в случае выбора этого пункта. Если атрибут не указан, серверная программа получит текст выбранного пункта;
- `selected` — атрибут без значения, делающий этот пункт изначально выбранным;
- `disabled` — атрибут без значения, делающий пункт недоступным для выбора;
- `label` — сокращенная версия текста пункта, которая будет выводиться в раскрывающемся списке. (Честно говоря, непонятно, зачем она нужна...)

Код, приведенный в листинге 6.4, создает раскрывающийся список из четырех пунктов. Второй по счету пункт сделан изначально выбранным.

Листинг 6.4

```
<select name="category">
  <option value="1">HTML</option>
  <option value="2" selected>CSS</option>
  <option value="3">JavaScript</option>
  <option value="4">PHP</option>
</select>
```

А код из листинга 6.5 создает обычный список высотой в пять пунктов, в котором можно выбрать сразу несколько пунктов из представленных шести.

Листинг 6.5

```
<select name="windows" size="5" multiple>
  <option value="xp">Windows XP</option>
  <option value="vista">Windows Vista</option>
  <option value="7">Windows 7</option>
  <option value="8">Windows 8</option>
  <option value="81">Windows 8.1</option>
  <option value="10">Windows 10</option>
</select>
```

Серверная программа получит от списка значение выбранного в нем пункта. Например, если в списке, чей код приведен в листинге 6.4, выбрать третий пункт, будет отправлена пара `category=3`. Если значение пункта не указано (атрибут `value` отсутствует в теге `<option>`), вместо него будет отправлен текст пункта (например, `category=JavaScript`). Если же ни один пункт не был выбран, серверная программа не получит от списка никаких данных.

В случае если список позволяет выбрать сразу несколько пунктов, то будут отправлены значения всех выбранных пунктов. Так, при выборе второго и третьего пунктов в списке, код которого приведен в листинге 6.5, серверная программа получит пары `windows=vista` и `windows=7`.

HTML позволяет объединять пункты списка в группы по какому-либо родственному признаку. Эти группы представляют собой наборы пунктов с заголовками. Текст пунктов, входящих в группу, выводится с небольшим отступом слева (рис. 6.2).

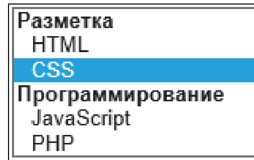


Рис. 6.2. Список, пункты которого разбиты на группы

Группу пунктов создают с помощью парного тега `<optgroup>` — в нем помещают теги, создающие пункты списка, которые входят в эту группу. Тег `<optgroup>` может присутствовать только внутри тега `<select>`.

Для указания параметров группы пунктов списка мы применим следующие атрибуты тегов:

- `label` — обязательный атрибут тега, задающий заголовок группы;
- `disabled` — атрибут без значения, делающий все пункты группы недоступными для выбора.

Код, приведенный в листинге 6.6, создает список из четырех пунктов, разбитых на две группы (этот список и показан на рис. 6.2).

Листинг 6.6

```
<select name="category" size="6">
  <optgroup label="Разметка">
    <option value="1">HTML</option>
    <option value="2" selected>CSS</option>
  </optgroup>
  <optgroup label="Программирование">
    <option value="3">JavaScript</option>
    <option value="4">PHP</option>
  </optgroup>
</select>
```

Поле ввода файла

Поле ввода файла отличается от всех прочих полей ввода, рассмотренных в этой главе. В нем значением, отправляемым серверной программе, выступает содержимое указанного посетителем файла (или файлов).

Поле ввода файла выводится на экран как обычное поле ввода, правее которого находится кнопка **Обзор**. При нажатии этой кнопки на экране появится стандартное диалоговое окно открытия файла, в котором посетитель сможет выбрать нужный файл или файлы.

Для создания поля ввода файла нужно задать атрибуту `type` тега `<input>` значение `file`:

```
<input type="file" name="image_file">
```

Для указания дополнительных параметров поля ввода файла применяются следующие необязательные атрибуты тега `<input>`:

- `accept` — задает типы файлов, доступные для выбора. Можно указать следующие значения:
  - *`<расширение файла с начальной точкой>`* — только файлы с указанным расширением;

- `image/*` — только графические файлы;
- `audio/*` — только аудиофайлы;
- `video/*` — только видеофайлы;
- *<MIME-тип>* — только файлы указанного MIME-типа. Все поддерживаемые MIME-типы и их описания можно найти по интернет-адресу **http://www.iana.org/assignments/media-types/**.

Если атрибут не указан, посетитель может выбрать файл любого типа;

- `multiple` — атрибут без значения, указывающий, что посетитель может выбрать сразу несколько файлов (по умолчанию можно выбрать лишь один файл).

> **Внимание!**
>
> Чтобы Web-форма смогла успешно отправить файл серверной программе, для нее необходимо указать метод кодирования данных `multipart/form-data`.

Листинг 6.7 представляет форму с тремя полями ввода файла. В первом можно выбрать лишь графический файл формата GIF (его тип задан расширением), во втором — видеофайл любого формата, в третьем — лишь архив ZIP (указан MIME-типом).

Листинг 6.7

```
<form action="/apps/upload.php" method="post"
enctype="multipart/form-data">
  <p>Изображение GIF: <input type="file" name="image_file"
  accept=".gif"></p>
  <p>Видео: <input type="file" name="video_file" accept="video/*"></p>
  <p>Архив ZIP: <input type="file" name="zip_file"
  accept="application/zip"></p>
</form>
```

Скрытое поле

Собственно, *скрытое поле* HTML — это не элемент управления, поскольку оно не предназначено для ввода данных и, более того, вообще не отображается на странице. Скрытое поле используется для хранения каких-либо внутренних данных сайта: уникальных идентификаторов записей, каких-либо признаков и проч.

Чтобы создать скрытое поле, следует дать атрибуту `type` тега `<input>` значение `hidden`. И не забыть указать собственно значение, которое хранится в этом поле, использовав знакомый нам атрибут тега `value`:

```
<input type="hidden" name="article_id" value="87">
```

Кнопки

И напоследок мы узнаем, как создаются кнопки. Ведь нажатие кнопки отправки данных серверной программе — последняя операция, которую выполняет занесший в форму данные посетитель.

HTML поддерживает создание кнопок четырех различных типов. Тип кнопки указывается значением атрибута `type` тега `<input>`:

- `submit` — *кнопка отправки данных*. При нажатии на нее Web-форма, в которой находится эта кнопка, выполнит отправку данных серверной программе;
- `reset` — *кнопка очистки*. При нажатии она заносит в элементы управления изначальные данные, заданные для них в HTML-коде, или очищает их, если изначальные данные не были указаны;
- `button` — обычная кнопка. Служит для запуска различных Web-сценариев (которыми мы займемся в *разделе 3*);
- `image` — *графическая кнопка*. Отображается на странице в виде графического изображения и при нажатии выполняет отправку данных серверной программе, как и кнопка отправки данных.

Атрибут тега `value` задает надпись для кнопки. Если он не указан, надпись задает сам Web-обозреватель, и у разных программ эта надпись различна.

Графическая кнопка поддерживает атрибуты тега `src`, служащего для указания интернет-адреса файла с графическим изображением, и `alt`, который задает текст замены.

В листинге 6.8 приведен фрагмент кода формы, включающий кнопки отправки данных и очистки.

Листинг 6.8

```
<form action="/apps/register.php" method="post">
  . . .
  <p><input type="submit" name="submit" value="Зарегистрироваться">
  <input type="reset" name="reset" value="Очистить"></p>
</form>
```

Кнопка отправки данных перешлет серверной программе в качестве значения свою надпись. Так, в нашем случае серверная программа получит от этой кнопки пару `submit=Зарегистрироваться` (разумеется, строка `"Зарегистрироваться"` будет закодирована). Это можно использовать, чтобы определить на стороне сервера, какая кнопка была нажата (если в форме их несколько). Кнопка очистки и обычная кнопка вообще не отправляют данные на сервер.

Элементы оформления

Элементы оформления, как следует из названия, служат для оформления форм и элементов управления.

Надпись

HTML предусматривает особый тег для создания *надписи*, т. е. поясняющего текста для элемента управления. Если посетитель щелкнет мышью на надписи, элемент управления получит фокус ввода.

Надпись создается парным тегом `<label>`, в который заносится текст надписи. Обязательный атрибут тега `for` указывает имя элемента управления, к которому относится надпись, — это имя задается в знакомом нам по *главе 5* атрибуте `id` тега, создающего сам элемент управления.

Надпись представляет собой встроенный элемент страницы. Она отображается на странице как обычный текст:

```
<label for="name1">Имя: </label><input name="name1" id="name1">
```

> **ВНИМАНИЕ!**
>
> Имя, заданное в атрибуте тега `id`, используется для привязки надписи к элементу управления (а также для привязки стилей и доступа к элементу страницы из Web-сценариев — подробнее об этом мы узнаем в *разделах 2* и *3*). А имя, заданное атрибутом тега `name`, указывается в составе данных, отправляемых формой серверной программе. Не забываем об этом.

В надписях также допускается использование атрибутов тегов `accesskey` и `tabindex`.

Группа

Группа служит для объединения нескольких элементов управления, в которые заносятся данные, относящиеся к единому блоку. Визуально она представляет собой рамку, окружающую элементы управления и, возможно, имеющую заголовок, расположенный на ее верхней границе.

Группа создается парным тегом `<fieldset>`, в который помещается HTML-код, создающий элементы управления, входящие в группу:

```
<fieldset>
  <элементы управления, объединяемые в группу>
</fieldset>
```

Тег `<fieldset>` также поддерживает атрибут без значения `disabled`.

Заголовок группы формируется парным тегом `<legend>`, внутрь которого помещается сам текст заголовка. Этот тег должен присутствовать в начале содержимого тега `<fieldset>`. Вне тега `<fieldset>` использовать тег `<legend>` не допускается.

Группа представляет собой блочный элемент страницы.

В листинге 6.9 приведен код формы, в которой создается группа, объединяющая поля для ввода регистрационных данных.

Листинг 6.9

```
<form action="/apps/register.php" method="post">
  <fieldset>
    <legend>Регистрационные данные</legend>
    <p>Имя: <input type="text" name="name1"></p>
    <p>Фамилия: <input type="text" name="name2"></p>
  </fieldset>
  <p><input type="submit" name="submit" value="Зарегистрироваться"></p>
</form>
```

Фреймы

Фрейм можно рассматривать как своего рода отдельное окно Web-обозревателя, помещенное прямо на страницу. В этом «окне» можно открыть любую другую страницу, даже входящую в состав другого сайта.

Фрейм создается парным тегом `<iframe>`. Необязательный атрибут `src` этого тега указывает интернет-адрес страницы, которая должна отображаться во фрейме, — если этот атрибут отсутствует в теге, фрейм будет пуст. Никакое содержимое в теге `<iframe>` не указывается.

Необязательный атрибут тега `id` может использоваться для задания имени фрейма. Это имя впоследствии можно указать в атрибуте `target` тегов `<a>` или `<form>` — в таком случае страница, на которую указывает гиперссылка или которая будет сформирована серверной программой в результате обработки данных, будет открыта в этом фрейме.

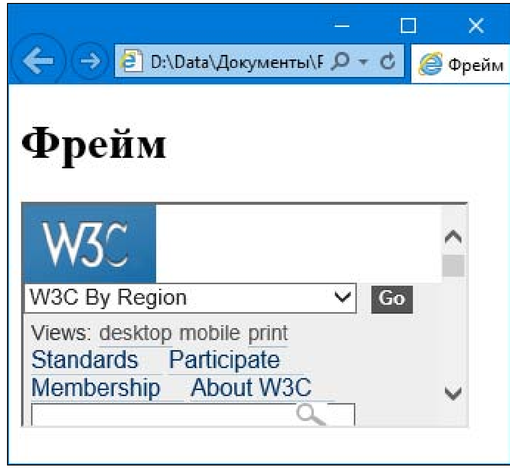


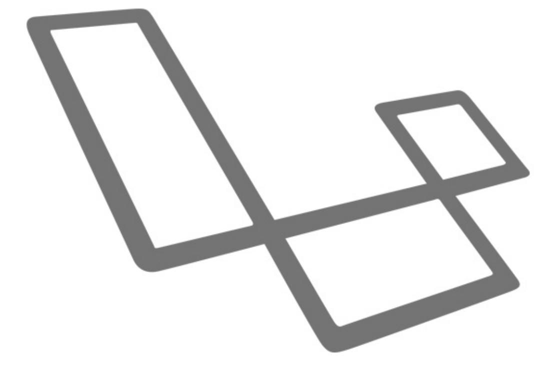
Рис. 6.3. Фрейм

Фрейм представляет собой блочный элемент страницы. Визуально он выводится в виде прямоугольника с довольно толстой рамкой, в котором и отображается страница с указанным интернет-адресом (рис. 6.3).

Этот фрейм, в котором выведена главная страница сайта организации W3C, формируется следующим кодом:

```
<h1>Фрейм</h1>
<iframe src="http://www.w3c.org/"></iframe>
```

Фреймы часто применяются, чтобы скрыть от посетителя какую-либо служебную или не предназначенную для посторонних информацию, выводящуюся серверной программой. Для этого фрейм делается невидимым (как этого добиться, будет рассказано в *главе 10*).

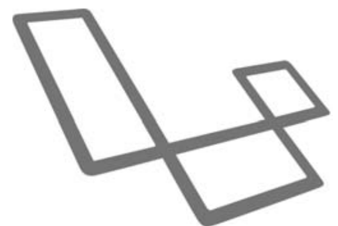


I. РАЗДЕЛ 2

Представление Web-страниц. Каскадные таблицы стилей CSS 3

Глава 7.	Введение в CSS
Глава 8.	Селекторы стилей. Единицы измерения CSS
Глава 9.	Параметры текста
Глава 10.	Отображение и видимость элементов. Параметры курсора. Генерируемое содержание
Глава 11.	Параметры фона. Градиентные фоны CSS 3
Глава 12.	Размеры, отступы, рамки, тени и выделение. Параметры таблиц
Глава 13.	Инструменты для создания разметки
Глава 14.	Специальные эффекты CSS 3
Глава 15.	Медиазапросы. Управление выводом на печать

ГЛАВА 7



Введение в CSS

Предыдущий раздел книги был посвящен содержанию Web-страниц и языку HTML, на котором они создаются. Мы изучили много терминов и тегов языка HTML и теперь сможем создать страницу, включающую текст, таблицы, графику, гиперссылки и даже аудио- и видеоролики.

Только вот выглядят эти страницы как-то невзрачно. Однообразный текст, похожие друг на друга абзацы, таблицы без рамок, тоскливая черно-белая расцветка... Не помешает их как-то оформить. Вы так не считаете?..

За оформление страниц и отдельных их элементов «отвечает» *представление*. Именно представление поможет нам оформить абзацы, таблицы и гиперссылки так, как хотим мы, а не Web-обозреватель (точнее, его разработчики). И именно представление поможет нам сделать страницы привлекательными.

Понятие о стилях CSS

Для создания представления страниц предназначена технология *каскадных таблиц стилей* (Cascading Style Sheets, CSS), или просто *таблиц стилей*. Таблица стилей содержит набор правил (*стилей*), описывающих оформление самой страницы и отдельных ее фрагментов. Эти правила определяют цвет текста и выравнивание абзаца, отступы между блоками, наличие и параметры рамки у таблицы, цвет фона страницы и многое другое.

Таблицы стилей пишут на особом языке, который так и называется — CSS. На данный момент самой современной версией этого языка является CSS 3, ее разработка все еще ведется, но многие Web-обозреватели уже поддерживают большую часть возможностей, описанных в ее черновой спецификации.

Именно CSS 3, точнее, его подмножество, поддерживаемое всеми современными Web-обозревателями, мы и будем изучать в этой книге.

Создание стилей CSS

Формат определения стиля CSS представлен в листинге 7.1.

Листинг 7.1

```
<селектор> {
  <атрибут стиля 1>: <значение 1>;
  <атрибут стиля 2>: <значение 2>;
  . . .
  <атрибут стиля n-1>: <значение n-1>;
  <атрибут стиля n>: <значение n>;
}
```

Рассмотрим правила написания, или, как говорят Web-верстальщики, определения стиля:

- определение каждого стиля представляет собой селектор, за которым через пробел в фигурных скобках следует набор атрибутов стилей и их значений;
- *селектор* служит двум целям. Во-первых, он указывает на элементы страницы, на которые должен действовать стиль, задавая для них оформление. Во-вторых, селектор однозначно идентифицирует сам стиль;
- *атрибут стиля* (не путать с атрибутом тега!) представляет один из параметров элемента страницы: цвет шрифта, выравнивание текста, величину отступа, толщину рамки и др. Каждый поддерживаемый CSS атрибут стиля имеет уникальное имя, однозначно идентифицирующее его;
- *значение* атрибута стиля указывает конкретное значение соответствующего параметра. Оно ставится после собственно атрибута стиля через символ `:` (двоеточие). В некоторых случаях значение атрибута стиля заключается в двойные кавычки;
- пары `<атрибут стиля>:<значение>` отделяются друг от друга символом `;` (точка с запятой). Между последней парой `<атрибут стиля>:<значение>` и закрывающей фигурной скобкой символ `;` можно не ставить, но обычно его все же указывают;
- для удобства чтения отдельные пары `<атрибут стиля>:<значение>` располагаются на отдельных строках. Также допускается форматировать их, вставляя между именами атрибутов стилей и их значениями пробелы.

Вот несколько примеров стилей:

- `p { color: #0000ff; }`

  Начнем рассмотрение этого стиля с атрибута стиля `color`. Он задает цвет текста. А его значение — `#0000ff` — представляет код синего цвета, записанный в формате RGB. (Об RGB-кодах и вообще способах задания цвета мы поговорим в *главе 8*.)

  Селектором у этого простого стиля выступает имя тега `<p>` (говорят, что такой селектор включает в свой состав всего один указатель). Следовательно, приве-

денный стиль будет автоматически применен Web-обозревателем ко всем абзацам, и все абзацы на странице будут набраны синим текстом;

- ```
  p.green { color: #00ff00;
            font-weight: bold; }
  ```

  Значение `#00ff00` атрибута стиля `color` задает зеленый цвет. Атрибут стиля `font-weight` устанавливает степень «жирности» шрифта, а его значение `bold` — полужирный шрифт.

  Что касается селектора, то он также содержит один указатель — комбинацию имени тега `<p>` и имени стилевого класса `green`. Этот стиль будет применен к абзацам, к которым привязан упомянутый стилевой класс. Привязка стилевого класса к элементу страницы выполняется атрибутом тега `class`:

  ```
  Этот абзац будет выведен зеленым полужирным
  шрифтом
  ```

- еще стиль:

  ```
  p .green { color: #00ff00;
             font-weight: bold; }
  ```

  А здесь селектор содержит уже два указателя: имя тега `<p>` и имя стилевого класса `green` — это именно два указателя, т. к. их разделяет пробел. Такой стиль будет применен к элементу страницы, к которому привязан стилевой класс `green` и который вложен в абзац:

  ```
  Этот фрагмент абзаца будет набран зеленым
  полужирным шрифтом, а этот текст - обычным шрифтом
  ```

- последний стиль:

  ```
  div#main p:first-child { font-size: larger; }
  ```

  Атрибут стиля `font-size` устанавливает размер шрифта. Его значение `larger` задает его увеличенный размер.

  Селектор этого стиля также включает два указателя, в качестве которых выступают комбинация имени тега `<div>` и имени элемента страницы `main` и комбинация имени тега `p` и псевдокласса `:first-child`. И стиль этот будет применен к абзацу, являющемуся первым потомком блока с именем `main`:

  ```

    Этот абзац будет выведен увеличенным шрифтом
    А этот - обычным шрифтом

  ```

Как видим, селектор позволяет точно указать, к какому элементу страницы должен быть привязан стиль. Принципы написания селекторов и разновидности применяемых в них указателей мы во всех подробностях рассмотрим в *главе 8*.
  ```

Таблицы стилей. Встроенные стили

Все стили, задающие представление страницы, нужно где-то хранить. Хранить их можно двумя способами.

Во-первых, стили могут быть помещены в *таблицу стилей*, которую можно рассматривать как набор стилей, представляющий собой отдельную от HTML-кода структуру. Такие таблицы стилей бывают двух видов: внешние и внутренние.

Внешние таблицы стилей хранятся отдельно от Web-страниц, в файлах с расширением css. Они содержат только CSS-код определений стилей.

В листинге 7.2 приведен пример внешней таблицы стилей.

Листинг 7.2

```
p { color: #0000ff; }
p .green { color: #00ff00;
           font-weight: bold; }
div#main h1:first-child { font-size: larger; }
```

Если внешняя таблица стилей хранится в отдельном файле, значит, нужно как-то привязать ее к странице. Для этого предназначен одинарный метатег `<link>`, который обычно помещается в секцию заголовка соответствующей страницы. (О метатегах и секциях страницы говорилось в *главе 1*.) Вот формат его написания:

```
<link rel="stylesheet" href="<интернет-адрес файла таблицы стилей>"
type="text/css">
```

Интернет-адрес файла таблицы стилей указывается в качестве значения обязательного атрибута `href` этого тега.

Остальные атрибуты тега `<link>` носят чисто служебный характер. Атрибут `rel` указывает, чем является файл, на который ссылается тег `<link>`, для текущей страницы. Его значение `stylesheet` говорит, что этот файл — внешняя таблица стилей. А атрибут `type` указывает тип MIME файла, на который ссылается этот тег, — внешняя таблица стилей имеет MIME-тип `text/css`.

Рассмотрим пример привязки таблицы стилей, хранящейся в файле `main.css` в корневой папке сайта:

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

Преимущество внешних таблиц стилей в том, что их можно привязать сразу к нескольким Web-страницам, задав тем самым для них одинаковое представление. Недостаток всего один, да и тот несущественный, — внешняя таблица стилей хранится в отдельном файле, который Web-обозревателю придется загружать отдельно, что может породить небольшую задержку.

Внутренняя таблица стилей записывается прямо в HTML-коде страницы. Ее заключают в парный тег `<style>`, который также, как правило, помещают в секцию заголовка. В остальном она не отличается от ее внешней «коллеги» (листинг 7.3).

Листинг 7.3

```
<head>
  . . .
  <style>
    p { color: #0000ff }
    p .green { color: #00ff00;
               font-weight: bold; }
    div#main h1:first-child { font-size: larger; }
  </style>
  . . .
</head>
```

Внутренняя таблица стилей является неотъемлемой частью страницы и, стало быть, загружается вместе с ней без всякой задержки. Однако стили, определенные во внутренней таблице стилей, применяются только к той странице, в которой эта таблица стилей находится. Следовательно, если мы захотим задать одинаковое представление для нескольких страниц, нам придется дублировать таблицу стилей в каждой из них.

В одной и той же Web-странице могут присутствовать сразу несколько таблиц стилей: несколько внешних и внутренняя (листинг 7.4).

Листинг 7.4

```
<head>
  . . .
  <link rel="stylesheet" href="/styles1.css" type="text/css">
  <link rel="stylesheet" href="/styles2.css" type="text/css">
  . . .
  <style>
    . . .
  </style>
  . . .
</head>
```

В таком случае действие всех этих таблиц стилей складывается. А по каким правилам — мы скоро выясним.

Во-вторых, стиль можно поместить прямо в тег, к которому он должен быть применен, создав тем самым *встроенный стиль*. Такой стиль представляет собой лишь набор атрибутов и их значений, разделенных символами точки с запятой. Селектор и фигурные скобки при этом не указываются, поскольку они просто не нужны.

Встроенный стиль помещается в тег в качестве значения особого атрибута `style` этого тега:

```
<p><span style="color: #ff0000;">Красный</span> текст</p>
```

В результате слово «Красный» в тексте страницы будет набрано красным цветом (его обозначает RGB-код `#ff0000`).

Встроенные стили применяются довольно редко, в основном, в экспериментальных целях и для указания стиля, который должен быть применен всего к одному элементу страницы. В остальных случаях много удобнее использовать полноценные стили с селекторами, поместив их во внешнюю таблицу стилей.

Правила каскадности и приоритет стилей

В Web-страницах со сложным представлением может случиться так, что на один и тот же элемент оказывают влияние сразу несколько стилей. Это могут быть стили с разными селекторами или определенные в разных таблицах стилей. И практически всегда эти стили задают для элемента разные представления, из-за чего возникает так называемый *конфликт стилей*.

Но как Web-обозреватель его разрешает? Из какого стиля он берет представление для такого элемента страницы? А может быть, он как-то объединяет все стили и применяет к элементу результат этого объединения? Давайте разберемся.

Создадим для примера внешнюю таблицу стилей (листинг 7.5).

Листинг 7.5

```
p { color: #eeeeee;
    text-align: justify; }
```

Ее единственный стиль задает для абзацев темно-серый цвет (RGB-код `#eeeeee`) и полное выравнивание текста (значение `justify` атрибута стиля `text-align`).

После этого мы изготовим страницу, содержащую внутреннюю таблицу стилей, чей код приведен в листинге 7.6.

Листинг 7.6

```
<style>
  p { color: #000000; }
  p.red { color: #ff0000; }
</style>
```

RGB-код `#000000` задает черный цвет.

А в секции тела страницы напишем HTML-код, показанный в листинге 7.7.

Листинг 7.7

```
<p>Это абзац</p>
<p class="red">Это другой абзац</p>
<p style="text-align: left;">Это третий абзац</p>
```

Атрибут тега `class`, как мы помним из начала этой главы, указывает имя стилевого класса для тега, в котором он присутствует.

Что же мы получим в результате?

Сначала Web-обозреватель загрузит внешнюю таблицу стилей и сохранит ее в памяти. После чего настанет черед внутренней таблицы стилей и встроенного стиля. (Отметим, что стили загружаются и обрабатываются именно в таком порядке.)

Далее он увидит, что и во внешней таблице стилей, и во внутренней присутствует один и тот же стиль — с селектором `p`. Внутренняя таблица стилей имеет больший приоритет, поскольку присутствует непосредственно в коде страницы. И Web-обозреватель объединит действие обоих стилей, перекрыв значения атрибутов стиля из внешней таблицы:

```
p { color: #eeeeee;
    text-align: justify; }
```

значениями одноименных атрибутов стиля из внутренней:

```
p { color: #000000; }
```

В итоге к первому абзацу нашей страницы будет применен следующий результирующий стиль:

```
p { color: #000000;
    text-align: justify; }
```

Далее Web-обозреватель увидит стиль с селектором `p.red`:

```
p.red { color: #ff0000; }
```

Он применяется только к абзацам с привязанным стилевым классом `red`, т. е. к меньшему числу элементов страницы. Напротив, стиль с селектором `p` применяется к большему числу элементов — ко всем абзацам без исключения. А стиль, применяемый к меньшему числу элементов, согласно правилам CSS, имеет больший приоритет — со всеми вытекающими последствиями.

Тогда ко второму абзацу будет применен вот такой результирующий стиль:

```
p.red { color: #ff0000;
        text-align: justify; }
```

В третьем абзаце мы указали встроенный стиль. Такой стиль имеет максимальный приоритет, и значения его атрибутов будут перекрывать значения одноименных атрибутов всех стилей, что уже действуют на этот элемент.

В результате к третьему абзацу будет применен результирующий стиль такого вида:

```
{ color: #000000;
  text-align: left: }
```

А теперь подытожим, сведя воедино все правила, описывающие поведение Web-обозревателя при формировании окончательных стилей. Их еще называют *правилами каскадности*:

- внешняя таблица стилей, привязанная к странице позже, имеет больший приоритет, чем привязанная к странице раньше;
- внутренняя таблица стилей имеет больший приоритет, чем все внешние;
- стили, действующие на меньшее количество элементов (более конкретные, если так можно сказать), имеют больший приоритет, чем те, что действуют на большее количество элементов (менее конкретные);
- стили с одинаковым селектором, определенные в таблице стилей позже, имеют больший приоритет, чем определенные раньше;
- встроенные стили имеют максимальный приоритет.

В следующей главе, рассматривая способы написания селекторов стилей, мы дополним эти правила.

Наследование атрибутов стилей

И напоследок рассмотрим еще один важный вопрос. Он касается наследования атрибутов стилей и их значений.

Предположим, что мы написали код, представленный в листинге 7.8.

Листинг 7.8

```
div { font-size: 10pt; }
p   { color: #ff0000; }
. . .
<div>Это абзац.</div>
<div><p>Это тоже абзац.</p></div>
```

Здесь мы задали для блоков (тегов `<div>`) размер шрифта, равный 10 пунктам, а для абзацев — красный цвет текста.

Текст, помещенный непосредственно в блок (верхняя строка HTML-кода в листинге 7.8), будет выведен шрифтом с размером в 10 пунктов. А какой размер будет иметь шрифт абзаца, вложенного в блок (вторая строка HTML-кода в том же листинге)?

Тоже 10 пунктов. Дело в том, что любой элемент страницы наследует все параметры, не определенные явно в примененных к нему стилях, от своего родителя. Так, второй абзац в нашем примере унаследовал параметр размера шрифта, который мы не задали явно, от своего родителя — блока. Родитель элемента, в свою очередь, наследует параметры от своего родителя и т. д.

Эта особенность CSS, называемая *наследованием*, позволяет нам не задавать весь набор параметров для каждой группы элементов, которые мы собираемся создать

на странице, а указать их всего один раз — у их общего родителя. Так, мы можем задать общие для всех элементов параметры (например, цвет текста и шрифт) для страницы (тега `<body>`) или даже для самого документа (тега `<html>`) — и они будут унаследованы всеми их потомками.

Если же какой-то параметр не был указан для всех родителей элемента, включая и саму страницу, для него задается некое значение по умолчанию. Для большинства атрибутов стиля это значение строго определено, однако для некоторой их части установка значения по умолчанию отдается на откуп разработчикам программ Web-обозревателей. В дальнейшем, знакомясь с различными атрибутами стилей, мы узнаем значения по умолчанию, указанные для них стандартом CSS.

Практически все атрибуты стилей поддерживают два предопределенных значения:

- `inherit` — явно указывает, что значение этого атрибута стиля должно наследоваться от родителя;
- `initial` — явно задает для атрибута стиля его значение по умолчанию.

Эти значения используются в стилях очень редко, но нам обязательно нужно о них знать.

> ***Внимание!***
>
> Здесь нужно отметить, что не все атрибуты стилей наследуют свои значения от родителей элемента. В частности, это касается атрибутов стилей, задающих параметры преобразований и анимации CSS 3 (см. *главу 14*). В дальнейшем мы обязательно отметим, какие атрибуты стилей не поддерживают наследование.

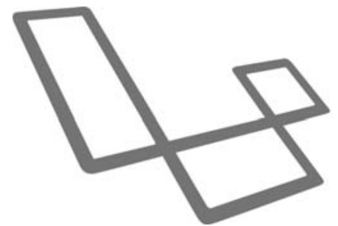
Комментарии CSS

В *главе 2* мы узнали о комментариях — особых фрагментах HTML-кода, которые не обрабатываются Web-обозревателем и служат для того, чтобы Web-верстальщик смог оставить какие-то заметки для себя или своих коллег. Для этого язык HTML предоставляет специальный тег.

Создавать комментарии позволяет и язык CSS. Это делается с помощью последовательностей символов `/*` и `*/`, между которыми помещают строки, предназначенные стать комментарием (листинг 7.9).

Листинг 7.9

```
/* Это комментарий CSS */
/*
  Это тоже комментарий CSS
*/
p { color: #0000FF; }
```



ГЛАВА 8

Селекторы стилей. Единицы измерения CSS

В предыдущей главе мы рассмотрели основы языка CSS, на котором пишутся стили, разновидности таблиц стилей и правила каскадности. В общем, положили начало знакомству с представлением страниц и процессом его создания.

В этой главе мы будем говорить о самом языке CSS. И начнем этот разговор с рассмотрения селекторов, указывающих элемент или группу элементов страницы, к которым должны быть применены стили.

Селекторы стилей

Как уже говорилось, селектор стиля указывает, к каким элементам страницы будет применен этот стиль. Web-обозреватель отыскивает элементы, совпадающие с селектором, и применяет к ним стиль автоматически.

Введение в селекторы стилей

Селектор стиля представляет собой набор указателей, отделенных друг от друга разделителями:

- *указатель* представляет элемент страницы, обладающий определенными признаками (например, созданный определенным тегом или имеющий привязанный к нему определенный стилевой класс);
- *разделитель* задает отношение элемента, описанного следующим за разделителем указателем, к элементу, описанному предыдущим указателем (скажем, он должен являться потомком этого элемента или располагаться следом за ним на том же уровне вложенности).

Первый указатель в селекторе задает начальный элемент страницы. Все последующие указатели представляют элементы, являющиеся либо его потомками, либо его «соседями» по тому же родительскому элементу. Стиль — автор подчеркивает это — всегда применяется к элементу, заданному последним указателем в селекторе.

Чтобы проиллюстрировать и закрепить полученные знания, рассмотрим несколько примеров.

- `div p em { . . . }`

  Здесь мы видим в селекторе три указателя — `div`, `p` и `em`, — отделенные друг от друга разделителями-пробелами. Все три указателя представляют собой имена тегов. Этот стиль будет применен к важному тексту (тегу `<em>`), находящемуся внутри абзаца (тега `<p>`), что, в свою очередь, располагается в блоке (теге `<div>`).

- `div#main > h1 { . . . }`

  А в этом селекторе всего два указателя: первый указывает на блок с именем `main`, а второй — на заголовок первого уровня (тег `<h1>`). В качестве разделителя здесь используется символ `>`, говорящий Web-обозревателю, что второй элемент (заголовок) должен быть непосредственно вложен в первый (блок). В результате стиль будет применен к заголовку первого уровня, являющемуся непосредственным потомком блока с именем `main`.

- `header .site-header h2::first-letter { . . . }`

  Первым указателем здесь выступает тег семантической «шапки» (`<header>`), вторым — элемент с привязанным стилевым классом `site-header`, а третьим — заголовок второго уровня (тег `<h2>`) с псевдоэлементом `::first-letter`, указывающим на первый символ его содержимого. Web-обозреватель применит этот стиль к первому символу содержимого заголовка второго уровня, вложенного в элемент с привязанным стилевым классом `site-header`, который является потомком семантической «шапки».

В определении стиля можно указать сразу несколько селекторов, перечислив их через запятую. В таком случае стиль будет применен к нескольким группам элементов, на которые указывают перечисленные в определении стиля селекторы:

```
p, blockquote, div.spoiler { . . . }
```

Этот стиль будет применен к абзацам, блочным цитатам (тегам `<blockquote>`) и блокам с привязанным стилевым классом `spoiler`.

Компоненты указателей

Язык CSS позволяет применять в селекторах как простые указатели, включающие в себя лишь один компонент, так и составные, содержащие несколько компонентов. Компоненты указателей и их разновидности — настолько важная тема, что ей следует посвятить отдельный разговор.

Основные указатели

Основные указатели указывают на нужный элемент по имени формирующего его тега, имени самого элемента, имени привязанного к нему стилевого класса или их комбинации.

Далее приведены все типы основных указателей, поддерживаемых CSS.

- Имена тегов, записанные без символов `<` и `>`, — указывают на элементы, созданные соответствующими тегами:

```
p { text-align: justify; }
. . .
<p>Текст, выровненный по ширине</p>
<div>А этот текст не будет выровнен по ширине</div>
```

Как видим, этот стиль применяется ко всем абзацам.

```
p em { color: #ff0000; }
. . .
<p><em>Красный важный</em> текст</p>
<h1><em>Важный</em>, но не красный текст</h1>
```

Этот стиль будет применен к любому важному тексту, вложенному в абзац.

- Имена элементов страницы, заданные с помощью атрибута тега `id`, — указывают на элементы, для которых заданы эти имена.

  Отметим, что в селекторе имя элемента указывается с начальным символом «решетки» (`#`), а в атрибуте тега `id` — без него.

```
#main { background-color: #ffffff; }
. . .
<div id="main">Этот блок будет иметь белый фон</div>
```

  Этот стиль будет применен к элементу с именем `main`; атрибут стиля `background-color` задает цвет фона элемента, а его значение `#ffffff` — белый цвет.

- Имена стилевых классов — указывают на элементы, к которым были привязаны эти стилевые классы.

  Стилевой класс может быть привязан к любому тегу путем указания его имени в атрибуте `class` этого тега. Отметим, что в селекторе стиля имя стилевого класса указывается с начальным символом точки, а в атрибуте тега `class` — без него:

```
.red { color: #ff0000; }
. . .
<p><span class="red">Красный</span> текст</p>
```

  В атрибуте тега `class` можно указать имена нескольких стилевых классов, записав их через пробелы. При этом стилевые классы, указанные позже, будут иметь больший приоритет, чем указанные раньше.

- Комбинации упомянутых ранее элементов, которые должны быть записаны без пробелов:

```
span.red { color: #ff0000; }
. . .
<p><span class="red">Красный</span> текст</p>
<p class="red">А этот текст уже не будет красным</p>
```

  Этот стиль будет применен лишь к вложенным контейнерам (тегам `<span>`), к которым привязан стилевой класс `red`.

```
p.highlighted span { color: #ff0000 }
. . .
```

```
<p class="highlighted"><span>Красный</span> текст</p>
<p><span class="highlighted">А этот текст</span> уже не будет
красным</p>
```

А этот стиль будет применен лишь к вложенным контейнерам — потомкам абзацев с привязанным стилевым классом `highlighted`.

- *Универсальный указатель* `*` — указывает на все элементы страницы, созданные с применением любых тегов:

```
blockquote * { color: #eeeeee; }
. . .
<blockquote>
  <p>Темно-серый текст</p>
  <div>Тоже темно-серый текст</div>
</blockquote>
<p>А этот текст не будет темно-серым</p>
```

Как видим, этот стиль применяется к любому элементу, вложенному в блочную цитату.

Указатели на атрибуты тега

Указатели на атрибуты тега позволяют указать на элемент страницы, тег которого содержит определенный атрибут или атрибут, имеющий определенное значение.

В случае использования с основным указателем указатель на атрибут тега записывается после него, пробелы между ними не допускаются.

Давайте рассмотрим все поддерживаемые CSS указатели на атрибуты тега:

- `[<имя атрибута тега>]` — указывает на элементы, теги которых содержат атрибут с заданным *именем*;
- `[<имя атрибута тега>=<значение>]` — указывает на элементы, у которых заданный *атрибут тега* имеет заданное *значение*;
- `[<имя атрибута тега>~=<слово>]` — указывает на элементы, значение *атрибута тега* которых включает заданное *слово*. «Словом» в CSS считается фрагмент значения, ограниченный пробелами или дефисами;
- `[<имя атрибута тега>*=<подстрока>]` — указывает на элементы, значение *атрибута тега* которых включает *подстроку*. Такая подстрока не обязательно должна представлять собой целое слово и может быть его фрагментом;
- `[<имя атрибута тега>|=<слово>]` — указывает на элементы, значение *атрибута тега* которых начинается с заданного *слова*;
- `[<имя атрибута тега>^=<подстрока>]` — указывает на элементы, значение *атрибута тега* которых начинается с заданной *подстроки*;
- `[<имя атрибута тега>$=<подстрока>]` — указывает на элементы, значение *атрибута тега* которых заканчивается заданной *подстрокой*.

Рассмотрим теперь несколько примеров:

- `img[alt] { . . . }`

  Этот стиль будет применен только к изображениям (тегам `<img>`), для которых задан текст замены (атрибут тега `alt`);

- `img[alt~=Пример] { . . . }`

  Этот стиль будет применен к изображениям, текст замены которых начинается со слова «Пример»: «Пример изображения», «Пример аудиофайла» и т. п., но не «Примерка»;

- `img[src^=/thumbnails/] { . . . }`

  Этот стиль будет применен к изображениям, чье содержимое — графические файлы — загружается с интернет-адресов, начинающихся с подстроки `/thumbnails/` (т. е. файлы, хранящиеся в папке thumbnails корневой папки сайта);

- `img[src$=.svg] { . . . }`

  А этот стиль будет применен к изображениям, в которых выводится содержимое графических файлов формата SVG.

Псевдоклассы

Псевдоклассы — это особая разновидность компонентов указателей, задающая либо местоположение элемента страницы в его родителе (скажем, является ли он первым или последним потомком), либо состояние этого элемента (например, если это элемент управления, то доступен ли он).

В случае использования в составе комбинации указателей, псевдокласс записывается после основного указателя или указателя на атрибут тега (если он присутствует) без всяких пробелов.

Язык CSS поддерживает довольно много самых разных псевдоклассов. Далее приведены все эти псевдоклассы и элементы страницы, на которые они указывают.

- `:empty` — пустые элементы, не содержащие никакого содержимого:

  ```
  p:empty { display: none; }
  ```

  Этот стиль будет применен ко всем пустым абзацам, скрывая их. Значение `none` атрибута стиля `display` полностью скрывает элемент страницы.

- `:first-child` — элемент, созданный указанным тегом и являющийся первым элементом в родителе:

  ```
  p:first-child { font-size: larger; }
  . . .
  <div>
    <p>Этот абзац будет выведен увеличенным шрифтом.</p>
    <p>А этот будет выведен шрифтом стандартного размера.</p>
  </div>
  <div>
    <h1>Заголовок будет выведен шрифтом стандартного размера, поскольку
    он не является абзацем</h1>
  ```

```
  <p>Этот абзац также будет выведен шрифтом стандартного размера,
  поскольку он не является первым потомком своего
  предка.</p>
</div>
```

- `:first-of-type` — первый элемент, созданный указанным тегом, в родителе:

```
p:first-of-type { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
  <p>А этот будет выведен шрифтом стандартного размера.</p>
</div>
<div>
  <h1>Заголовок</h1>
  <p>Этот абзац будет выведен увеличенным шрифтом, поскольку он
  является первым абзацем — потомком своего предка.</p>
</div>
```

- `:last-child` — элемент, созданный указанным тегом и являющийся последним элементом в родителе.
- `:last-of-type` — последний элемент, созданный указанным тегом, в родителе.
- `:nth-child(<n>)` — элементы, созданные указанным тегом и являющиеся точно *n*-ми по счету элементами в родителе:

```
p:nth-child(2) { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>А этот будет выведен увеличенным шрифтом.</p>
</div>
<div>
  <h1>Заголовок</h1>
  <p>Этот абзац также будет выведен увеличенным шрифтом,
  поскольку он является вторым потомком своего предка.</p>
</div>
```

Аргумент, задающий порядковый номер потомка, может быть указан в виде формулы вида `<A>n + <B>`, где `A` и `B` — числа, а `n` последовательно принимает значения от `n` до количества элементов в родителе. В результате получится порядковый номер элемента, к которому будет применен стиль:

```
p:nth-child(3n + 1) { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
```

```
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
</div>
```

В качестве аргумента также могут быть использованы предопределенные значения `odd` и `even`, указывающие, соответственно, все нечетные и четные элементы в родителе:

```
p:nth-child(even) { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
</div>
```

- `:nth-last-child(<n>)` — элементы, созданные указанным тегом и являющиеся точно *n*-ми по счету элементами в родителе, если считать с конца.
- `:nth-of-type(<n>)` — *n*-е элементы, созданные указанным тегом, в родителе. Здесь для задания аргумента также могут использоваться формулы и значения `odd` и `even`:

```
p:nth-of-type(2) { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>А этот будет выведен увеличенным шрифтом.</p>
</div>
<div>
  <h1>Заголовок</h1>
  <p>Этот абзац будет выведен шрифтом стандартного размера.</p>
  <p>Этот абзац будет выведен увеличенным шрифтом,
  поскольку он является вторым абзацем — потомком своего предка.</p>
</div>
```

- `:nth-last-of-type(<n>)` — *n*-е элементы, созданные указанным тегом, в родителе, если считать с конца.
- `:only-child` — элемент, созданный указанным тегом и являющийся единственным элементом в родителе.

```
p:only-child { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
</div>
<div>
  <p>А этот будет выведен шрифтом стандартного размера.</p>
  <p>И этот тоже.</p>
</div>
```

```
<div>
  <h1>Заголовок</h1>
  <p>И этот абзац будет выведен шрифтом стандартного размера,
  поскольку он не единственный потомок своего предка.</p>
</div>
```

- `:only-of-type` — единственный элемент, созданный указанным тегом, в родителе.

```
p:only-of-type { font-size: larger; }
. . .
<div>
  <p>Этот абзац будет выведен увеличенным шрифтом.</p>
</div>
<div>
  <p>А этот будет выведен шрифтом стандартного размера.</p>
  <p>И этот тоже.</p>
</div>
<div>
  <h1>Заголовок</h1>
  <p>Этот абзац будет выведен увеличенным шрифтом, поскольку он
  единственный абзац — потомок своего предка.</p>
</div>
```

- `:root` — корневой тег страницы, на практике то же самое, что и тег `<html>`.
- `:active` — активная гиперссылка, на которой посетитель в данный момент щелкает мышью.
- `:link` — непосещенная гиперссылка.
- `:visited` — посещенная гиперссылка.

  Псевдоклассы, относящиеся к гиперссылкам, используются только совместно с основным указателем, указывающим на тег `<a>`:

```
a:link { color: #0000ff; }
a:active, a:hover { color: #ff0000; }
a:visited { color: #00ff00; }
```

- `:target` — активный якорь (на который был выполнен переход щелчком на соответствующей гиперссылке). О якорях рассказывалось в *главе 5*.
- `:hover` — элемент, на который наведен курсор мыши.
- `:focus` — гиперссылка или элемент управления, имеющий фокус ввода (подробнее о фокусе ввода см. в *главе 5*).
- `:checked` — установленный флажок или переключатель (подробнее об элементах управления, поддерживаемых HTML, говорилось в *главе 6*).
- `:enabled` — доступный элемент управления.
- `:disabled` — недоступный элемент управления.

- `:required` — обязательный элемент управления (имеющий атрибут тега `required`).
- `:optional` — необязательный элемент управления (не имеющий атрибута тега `required`).
- `:valid` — элемент управления, в который занесено корректное значение.
- `:invalid` — элемент управления, в котором указано некорректное значение.

Псевдокласс `:not` стоит особняком. Он позволяет привязать стиль к любому элементу страницы, не удовлетворяющему заданным условиям. Вот формат записи этого псевдокласса:

```
<основной селектор>:not(<селектор выбора>)
```

Селектором выбора может выступать любой селектор, содержащий одинарный указатель (комбинации указателей не поддерживаются). В результате стиль будет привязан к элементу страницы, удовлетворяющему *основному селектору* и не удовлетворяющему *селектору выбора*.

Например, этот стиль будет применен лишь к абзацам, не являющимся первыми потомками своих родителей:

```
p:not(:first-child) { font-size: larger; }
```

Псевдоэлементы

Псевдоэлементы указывают на определенный фрагмент содержимого элемента.

Если псевдоэлемент используется в составе комбинации указателей, он добавляется к основному указателю, указателю на атрибут тега или псевдоклассу (если таковые имеются в наличии). Пробелы между ними не допускаются.

CSS 3 поддерживает всего два псевдоэлемента:

- `::first-letter` — указывает на первый символ содержимого элемента:

  ```
  p:first-of-type::first-letter { font-size: larger; }
  ```

  Здесь мы создаем буквицу у первого абзаца;
- `::first-line` — указывает на первую строку содержимого элемента.

Разделители

Разделители, как мы помним, задают отношение элемента, который описывается указателем, расположенным правее такого разделителя, к элементу, описанному указателем, что находится левее. С помощью разделителей мы можем указать, что следующий элемент должен являться потомком предыдущего, или оба этих элемента должны находиться на одном уровне вложенности и следовать друг за другом.

Разделителей в CSS не очень много — всего четыре:

- *пробел* — элемент, описываемый следующим указателем, должен быть вложен в элемент, описываемый предыдущим указателем, и необязательно непосредственно. С этим разделителем мы уже знакомы:

```
p em { color: #ff0000; }
. . .
<p><em>Этот текст</em> будет красным.</p>
<p><strong>И <em>этот текст</em> будет</strong> красным.</p>
<div>А <em>этот</em> - не будет.</div>
```

- `>` — элемент, описываемый следующим указателем, должен быть непосредственно вложен в элемент, описываемый предыдущим указателем, т. е. являться его прямым потомком:

```
p > em { color: #ff0000; }
. . .
<p><em>Этот текст</em> будет красным.</p>
<p><strong>А <em>этот текст</em> не будет</strong> красным.</p>
<div>И <em>этот</em> не будет.</div>
```

- `+` — элемент, описываемый следующим указателем, должен располагаться непосредственно за элементом, описываемым предыдущим указателем. Оба элемента должны иметь одинаковый уровень вложенности (т. е. быть потомками одного родителя):

```
h1 + p::first-letter { font-size: larger; }
. . .
<h1>Заголовок</h1>
<p>Этот абзац будет иметь буквицу.</p>
<p>А этот — не будет.</p>
```

- `~` — элемент, описываемый следующим указателем, должен располагаться за элементом, описываемым предыдущим указателем, и необязательно непосредственно. Оба элемента должны иметь одинаковый уровень вложенности:

```
h1 ~ p::first-letter { font-size: larger; }
. . .
<h1>Заголовок</h1>
<p>Этот абзац будет иметь буквицу.</p>
<h2>Другой заголовок</h2>
<p>И этот абзац будет иметь буквицу, т. к., в любом случае,
находится после заголовка первого уровня.</p>
```

Единицы измерения и вычисления CSS

Когда мы начнем создавать стили, нам придется указывать различные величины, такие как размер шрифта, толщину рамки, цвет, продолжительность анимации и проч. В каких единицах их указывать?

Скажем сразу, что обозначение нужной единицы ставится после собственно ее значения. Это относится ко всем единицам измерения, поддерживаемым CSS.

Единицы измерения размеров, используемые наиболее часто, представлены в табл. 8.1.

Таблица 8.1. *Единицы измерения размера, поддерживаемые стандартом CSS*

Название	**Обозначение в CSS**
Пикселы	`px`
Пункты	`pt`
Дюймы	`in`
Сантиметры	`cm`
Миллиметры	`mm`
Пики	`pc`
Размер текущего шрифта	`em`
Проценты от размера родительского элемента	`%`
1% от ширины клиентской области окна Web-обозревателя	`vw`
1% от высоты клиентской области окна Web-обозревателя	`vh`

Рассмотрим пару примеров:

- `p { font-size: 10pt; }`

  Здесь мы задаем размер шрифта для абзацев, равным 10 пунктам;
- `div#main { border-left-width: 1px; }`

  Задаем толщину левой стороны рамки у блока с именем `main`, равной одному пикселу.

Единицы измерения угла используются в записи преобразований, выполняемых с элементом страницы (подробнее — в *главе 14*). CSS поддерживает следующие единицы измерения углов:

- `deg` — градусы;
- `rad` — радианы;
- `grad` — грады;
- `turn` — обороты.

Единицы измерения времени понадобятся нам для создания анимации. Их всего две: `s` (секунды) и `ms` (миллисекунды).

Язык CSS 3 также предоставляет возможность указывать в качестве значения атрибута стиля результат какого-либо вычисления. Для этого служит функция `calc`. Само выражение, которое требуется вычислить, записывается сразу же после ее имени, берется в скобки и имеет вид обычной алгебраической формулы. В выражениях, вычисляемых в функции `calc`, допускается использовать следующие операции: сложение (обозначается символом `+`), вычитание (`-`), умножение (`*`) и деление (`/`), например:

```
div#main { width: calc(100wh — 400px); }
```

Этот стиль указывает ширину блока с именем `main`, равной ширине клиентской области окна Web-обозревателя за вычетом 400 пикселов.

Что касается значений цвета, то CSS поддерживает четыре способа их указания:

- *Шестнадцатеричный формат RGB.* Цвет записывается в виде:

  ```
  #<доля красного цвета><доля зеленого цвета><доля синего цвета>
  ```

  где каждая *доля* представляет собой шестнадцатеричное число от `00` до `ff`. Например, RGB-код `#000000` задает черный цвет, `#ff0000` — красный, а `#cccccc` — светло-серый.

- *Десятичный формат RGB.* Цвет записывается в виде:

  ```
  rgb(<доля красного цвета>,<доля зеленого цвета>,<доля синего цвета>)
  ```

  причем здесь каждая *доля* представляет собой десятичное число от `0` до `255`. Например, `rgb(255, 0, 0)` задает красный цвет.

- *Десятичный формат RGB с каналом прозрачности.* Здесь цвет указывается следующим образом:

  ```
  rgba(<доля красного цвета>,<доля зеленого цвета>,<доля синего цвета>,↳
  <прозрачность>)
  ```

  Прозрачность должна представлять собой число от `0.0` (цвет полностью прозрачен) до `1.0` (цвет полностью непрозрачен). Например, `rgba(255, 0, 0, 0.5)` задает полупрозрачный красный цвет.

- *Имя цвета.* Так, красный цвет имеет имя `Red`, а черный — `Black`. Все поддерживаемые имена цветов можно найти по интернет-адресу: **http://www.w3schools.com/colors/colors_names.asp**.

Важные атрибуты стилей

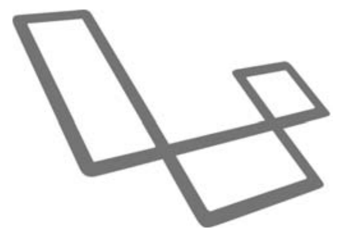
В *главе 7* мы изучили, помимо всего прочего, правила каскадности. Согласно им, Web-обозреватель формирует результирующие стили, которые потом и применяет к элементам страницы. И, согласно этим правилам, атрибуты стилей, имеющих больший приоритет или определенных в таблице стилей позже, перекрывают значения атрибутов стилей с меньшим приоритетом или определенных раньше.

Однако часто возникает необходимость нарушить это правило, указав для какого-либо атрибута определенного стиля максимальный приоритет. Тогда значение этого стиля не будет перекрыто никакими другими стилями, даже встроенными. Такой атрибут стиля носит название *важного*.

Важный атрибут стиля создается помещением между его значением и завершающим символом точки с запятой языковой конструкции `!important` (пишется слитно, без пробелов между восклицательным знаком и словом `important`). Вот так:

```
em { color: #00ff00 !important; }
```

Теперь текст, выделенный тегом `<em>`, всегда будет выводиться зеленым цветом, даже если этот параметр переопределен в стиле с более высоким приоритетом.



ГЛАВА 9

Параметры текста

Мы научились формировать селекторы стилей и познакомились с поддерживаемыми языком CSS единицами измерения. И теперь можем приступить к изучению атрибутов стилей, которые, собственно, и задают различные параметры элементов страниц.

Параметры шрифта

Начнем мы с атрибутов стилей, указывающих различные параметры шрифта, которым набран текст: название шрифта, размер, начертание и проч.

Атрибут стиля `font-family` задает имя шрифта, которым будет выведен текст:

```
font-family: <список имен шрифтов, разделенных запятыми>
```

Имена шрифтов задаются в виде их названий — например, Arial или Times New Roman. Если имя шрифта содержит пробелы, его нужно взять в кавычки. Если же имя шрифта указывается во встроенном стиле, то оно берется в апострофы:

```
p { font-family: Arial; }
h1 ( font-family: "Times New Roman"; }
<p style="font-family: 'Times New Roman';">
```

Если указанный нами шрифт присутствует на компьютере посетителя, Web-обозреватель его использует. Если же такого шрифта нет, то текст выводится шрифтом, выбранным разработчиками программы в качестве шрифта по умолчанию. И наша страница, возможно, будет выглядеть не так, как мы задумывали. (Впрочем, шрифты Arial и Times New Roman присутствуют на любом компьютере, работающем под управлением Windows.)

Можно указать несколько наименований шрифтов через запятую:

```
p { font-family: Verdana, Arial; }
```

Тогда Web-обозреватель сначала будет искать первый из указанных шрифтов, в случае неудачного поиска — второй, потом третий и т. д.

Вместо имени конкретного шрифта можно задать имя одного из *семейств шрифтов*, представляющих целые наборы аналогичных шрифтов. Таких семейств пять: serif (шрифты с засечками), sans-serif (шрифты без засечек), cursive (шрифты, имитирующие рукописный текст), fantasy (декоративные шрифты) и monospace (моноширинные шрифты):

```
p { font-family: Verdana, Arial, sans-serif; }
```

> ***Внимание!***
>
> Атрибут стиля `font-family`, равно как и все прочие атрибуты стилей, которые мы изучим впоследствии, также поддерживает предопределенные значения `initial` и `inherit`, о которых говорилось в *главе 7* и которые автор ради краткости не указал. Если какой-то из атрибутов стиля не поддерживает эти значения, мы отметим это явно.

Атрибут стиля `font-size` указывает размер шрифта:

```
font-size: <размер>|xx-small|x-small|small|medium|large|x-large|
xx-large|larger|smaller
```

Размер шрифта можно задать в любых единицах измерения размеров, поддерживаемых CSS и перечисленных в табл. 8.1. Например:

```
p { font-size: 10pt; }
strong { font-size: 1cm; }
em { font-size: 150%; }
```

Кроме числовых, атрибут `font-size` может принимать и символьные значения. Так, значения от `xx-small` до `xx-large` задают семь предопределенных размеров шрифта — от самого маленького до самого большого. А значения `larger` и `smaller` представляют следующий размер шрифта, соответственно, по возрастанию и убыванию. Например, если для родительского элемента определен шрифт размера `medium`, то значение `larger` установит для текущего элемента размер шрифта `large`.

Значение атрибута стиля `font-size` по умолчанию — `medium`. К сожалению, различные Web-обозреватели устанавливают его по-разному...

Атрибут стиля `color` задает цвет текста и горизонтальной линии HTML:

```
color: <цвет>
```

Зададим для текста красный цвет:

```
h1 { color: #ff0000; }
```

А теперь серый цвет:

```
address { color: #cccccc; }
```

По умолчанию все Web-обозреватели устанавливают в качестве цвета по умолчанию черный. Уж в этом-то их разработчики единодушны...

Атрибут стиля `opacity` позволяет указать степень полупрозрачности элемента страницы:

```
opacity: <числовое значение>
```

Значение полупрозрачности представляет собой число от `0.0` (элемент полностью прозрачен и, таким образом, невидим) до `1.0` (элемент полностью непрозрачен) — это значение по умолчанию.

Зададим половинную прозрачность для изображений:

```
img { opacity: 0.5; }
```

Атрибут стиля `font-weight` устанавливает «жирность» шрифта:

```
font-weight: normal|bold|bolder|lighter|100|200|300|400|500|600|
700|800|900
```

Здесь доступны семь абсолютных значений от `100` до `900`, представляющих различную «жирность» шрифта — от минимальной до максимальной, при этом обычный шрифт будет иметь «жирность» `400` (или `normal`), а полужирный — `700` (или `bold`). Значения `bolder` и `lighter` являются относительными и представляют следующие степени «жирности» соответственно в бóльшую и меньшую сторону. Значение по умолчанию — `normal`:

```
code { font-weight: bold; }
```

Атрибут `font-style` задает начертание шрифта:

```
font-style: normal|italic|oblique
```

Доступны три значения, представляющие обычный шрифт (`normal`), курсив (`italic`) и особое декоративное начертание, похожее на курсив (`oblique`).

Атрибут стиля `font-variant` позволяет указать, как будут выглядеть строчные буквы шрифта:

```
font-variant: normal|small-caps
```

Значение `small-caps` задает такое поведение шрифта, когда его строчные буквы выглядят точно так же, как прописные, просто имеют меньший размер. Значение `normal` задает для шрифта обычные прописные буквы (это, кстати, поведение по умолчанию).

Атрибут стиля `text-decoration` задает «украшение» (подчеркивание или зачеркивание), которое будет применено к тексту:

```
text-decoration: none|underline|overline|line-through
```

Здесь доступны четыре значения:

- `none` — убирает все «украшения», заданные для шрифта родительского элемента (значение по умолчанию);
- `underline` — подчеркивает текст;
- `overline` — «надчеркивает» текст, т. е. проводит линию над строками;
- `line-through` — зачеркивает текст.

> ***Внимание!***
>
> Не следует без особой нужды задавать для текста подчеркивание. Дело в том, что Web-обозреватели по умолчанию выводят гиперссылки подчеркнутыми, и подчеркнутый текст, не являющийся гиперссылкой, может обескуражить посетителя.

Атрибут стиля `text-transform` позволяет изменить регистр символов текста:

```
text-transform: capitalize|uppercase|lowercase|none
```

Мы можем преобразовать текст к верхнему (значение `uppercase` этого атрибута) или нижнему (`lowercase`) регистру, преобразовать к верхнему регистру первую букву каждого слова (`capitalize`) или оставить в изначальном виде (`none`) — это значение по умолчанию.

Атрибут стиля `line-height` задает высоту строки текста:

```
line-height: normal|<расстояние>
```

Здесь можно задать непосредственно значение высоты, использовав любую единицу измерения CSS (см. табл. 8.1). Если же мы не укажем единицу измерения, заданное нами значение будет умножено на высоту текущего шрифта и затем использовано. Таким образом, чтобы увеличить высоту строки вдвое по сравнению с обычной, мы можем написать:

```
p { line-height: 2; }
```

Значение `normal` этого атрибута возвращает управление высотой строки Web-обозревателю. Оно же является значением по умолчанию этого атрибута стиля.

Атрибут стиля `letter-spacing` позволяет задать дополнительное расстояние между символами текста:

```
letter-spacing: normal|<расстояние>
```

Отметим, что это именно дополнительное расстояние, и оно будет добавлено к изначальному, установленному самим Web-обозревателем. Расстояние может быть положительным и отрицательным — в последнем случае символы шрифта будут располагаться друг к другу ближе обычного. Значение `normal` устанавливает для дополнительного расстояния значение по умолчанию, равное нулю.

Вот пример задания для заголовков первого уровня дополнительного расстояния между символами, равного 5 пикселам:

```
h1 { letter-spacing: 5px; }
```

Текст, набранный такими символами, будет выглядеть разреженным (впрочем, заголовкам это пойдет на пользу).

А здесь мы задали отрицательное дополнительное расстояние между символами, равное двум пикселам:

```
h6 { letter-spacing: -2px; }
```

Эти два пиксела будут вычтены из изначального расстояния, в результате символы сблизятся, а текст станет выглядеть сжатым. Возможно, символы даже налезут друг на друга.

Атрибут стиля `word-spacing` задает дополнительное расстояние между отдельными словами текста:

```
word-spacing: normal|<расстояние>
```

Пример:

```
h1 { word-spacing: 5mm; }
```

Атрибут стиля `font` позволяет задать все основные параметры шрифта:

```
font: [<начертание>] [<вид строчных букв>] [<"жирность">]
[<размер>[/<высота строки текста>]] <имя шрифта>
```

Как видим, обязательным является только имя шрифта — остальные параметры могут отсутствовать.

Задаем для текста абзацев шрифт Times New Roman размером 10 пунктов:

```
p { font: 10pt "Times New Roman"; }
```

А для заголовков шестого уровня — шрифт Arial размером 12 пунктов и курсивного начертания:

```
h6 { font: italic 12pt Verdana; }
```

Параметры вывода текста

А теперь познакомимся с атрибутами стилей, управляющими выводом текста, в частности, указывающими выравнивание текста и отступ красной строки.

- Атрибут стиля `text-indent` задает отступ для красной строки:

  ```
  text-indent: <отступ красной строки>
  ```

  По умолчанию отступ красной строки равен нулю.

  Указываем для всех абзацев отступ красной строки в 5 мм:

  ```
  p { text-indent: 5mm; }
  ```

- Атрибут стиля `text-align` задает горизонтальное выравнивание текста:

  ```
  text-align: left|right|center|justify
  ```

  Здесь доступны значения `left` (выравнивание по левому краю — это обычное поведение Web-обозревателя), `right` (по правому краю), `center` (по центру) и `justify` (полное выравнивание) — например, так:

  ```
  p { text-align: justify; }
  h1 { text-align: center; }
  ```

- Атрибут стиля `vertical-align` задает вертикальное выравнивание текста:

  ```
  vertical-align: baseline|sub|super|top|text-top|middle|bottom|
  text-bottom|<промежуток между базовыми линиями>
  ```

  Этот атрибут стиля принимает восемь предопределенных значений:

  - `baseline` — выравнивает базовую линию фрагмента текста по базовой линии текста родительского элемента (поведение по умолчанию). *Базовой* называется воображаемая линия, на которой располагается текст;

- `super` — выравнивает базовую линию фрагмента текста по базовой линии верхнего индекса родительского элемента, создавая надстрочный индекс;
- `sub` — выравнивает базовую линию фрагмента текста по базовой линии нижнего индекса родительского элемента, создавая тем самым подстрочный индекс;
- `top` и `text-top` — выравнивает верхний край фрагмента текста по верхнему краю родительского элемента;
- `middle` — выравнивает центр фрагмента текста по центру родительского элемента;
- `bottom` и `text-bottom` — выравнивает нижний край фрагмента текста по нижнему краю родительского элемента.

Кроме того, мы можем указать для этого атрибута стиля абсолютное или относительное значение, задающее, насколько выше или ниже базовой линии текста родительского элемента должна находиться базовая линия фрагмента текста. Положительное значение смещает базовую линию фрагмента текста вверх, отрицательное — вниз.

Следующий стиль задает для текста расположение, совпадающее с базовой линией верхнего индекса, и уменьшенный размер шрифта. Фактически с помощью этого стиля мы превращаем текст в надстрочный индекс, выводимый согласно всем типографским правилам:

```
.super { vertical-align: super;
         font-size: smaller; }
```

Тот же атрибут стиля пригоден для выравнивания графических изображений, являющихся частью абзаца:

```
<p>Это картинка: <img style="vertical-align: text-bottom"
src="picture.png">.</p>
```

Этот HTML-код создает абзац с графическим изображением. Низ этого изображения будет выровнен по нижнему краю текста абзаца. Иными словами, изображение будет как бы возвышаться над текстом.

Скорее всего, для достижения нужного результата придется поэкспериментировать с различными значениями атрибута стиля `vertical-align`. Очень уж много у него возможных значений, и слишком разный они дают результат в различных случаях. Но Web-верстальщику не привыкать к экспериментам!

Параметры списков

Параметры списков включают в себя вид маркера или нумерации и положение маркера (нумерации) относительно текста пункта.

Атрибут стиля `list-style-type` задает вид маркеров или нумерации у пунктов списка:

```
list-style-type: none|disc|circle|square|decimal|decimal-leading-zero|
lower-alpha|lower-latin|lower-roman|upper-alpha|upper-latin|upper-roman
```

- `none` — маркер и нумерация отсутствуют (обычное поведение для не-списков);
- `disc` — маркер в виде черного кружка (обычное поведение для маркированных списков);
- `circle` — маркер в виде светлого кружка;
- `square` — маркер в виде квадратика. Он может быть светлым или темным, в зависимости от Web-обозревателя;
- `decimal` — нумерация арабскими цифрами (обычное поведение для нумерованных списков);
- `decimal-leading-zero` — нумерация арабскими цифрами от 01 до 99 с начальным нулем;
- `upper-alpha` и `upper-latin` — нумерация прописными латинскими буквами;
- `lower-alpha` и `lower-latin` — нумерация строчными латинскими буквами;
- `upper-roman` — нумерация римскими цифрами, набранными прописными буквами;
- `lower-roman` — нумерация римскими цифрами, набранными строчными буквами.

> ***На заметку***
>
> Атрибут стиля `list-style-type` поддерживает и другие значения, задающие нумерацию греческими, грузинскими и армянскими буквами, символами катаканы и хираганы. Список всех значений, поддерживаемых этим атрибутом стиля, можно увидеть на странице **http://www.w3schools.com/cssref/pr_list-style-type.asp**.

Атрибут стиля `list-style-type` можно задавать как для самих списков, так и для отдельных их пунктов. В последнем случае создается список, в котором пункты имеют разные маркеры или нумерацию. Иногда это может пригодиться.

Вот стиль для маркированного списка с маркером в виде квадратика:

```
ul { list-style-type: square; }
```

А в листинге 9.1 мы задали такой же маркер для одного из пунктов маркированного списка.

Листинг 9.1

```
.squared { list-style-type: square; }
. . .
<ul>
  <li>Первый пункт</li>
  <li class="squared">Второй пункт (с другим маркером)</li>
  <li>Третий пункт</li>
</ul>
```

Атрибут стиля `list-style-image` позволяет задать в качестве маркера пунктов списка какое-либо графическое изображение (*графический маркер*). В этом случае значение атрибута стиля `list-style-type`, если таковой задан, игнорируется:

```
list-style-image: none|url(<интернет-адрес файла изображения>)
```

Интернет-адрес файла изображения с графическим маркером берется в двойные кавычки:

```
ul { list-style-image: url("/markers/dot.gif"); }
```

Значение `none` убирает графический маркер и возвращает простой, неграфический. Это обычное поведение.

Атрибут стиля `list-style-image` также можно задавать как для самих списков, так и для отдельных их пунктов.

Атрибут стиля `list-style-position` позволяет указать местоположение маркера или нумерации в пункте списка:

```
list-style-position: inside|outside
```

Доступных значений здесь два:

- `inside` — маркер или нумерация находятся как бы внутри пункта списка, являются его частью;
- `outside` — маркер или нумерация вынесены за пределы пункта списка (это обычное поведение).

Вот пример:

```
ol { list-style-position: inside; }
```

CSS предоставляет нам также «всеобъемлющий» атрибут стиля `list-style`, который может задать сразу все параметры списка:

```
list-style: [<вид маркеров или нумерации>]
[<местоположение маркера или нумерации>] [<графический маркер>]
```

Например:

```
ol { list-style: lower-roman inside; }
```

Дополнительные параметры текста

Напоследок познакомимся с атрибутами стилей, устанавливающими дополнительные параметры текста: тень и режим вывода текста, а также позволяющими использовать на странице загружаемые шрифты.

Тень у текста

Любителям все украшать стандарт CSS 3 предлагает одну очень интересную возможность — создание тени у текста. При умеренном употреблении она может заметно оживить Web-страницу.

Параметры тени задает атрибут стиля `text-shadow`:

```
text-shadow: <горизонтальное смещение> <вертикальное смещение>
[<радиус размытия>] <цвет> | none
```

Горизонтальное смещение тени указывается в любой единице измерения, поддерживаемой CSS (см. табл. 8.1). Если задано положительное смещение, тень будет расположена правее текста, если отрицательное — левее. Можно также задать и нулевое смещение — тогда тень будет располагаться прямо под текстом. Нулевое смещение имеет смысл только в том случае, если для тени задано размытие.

Вертикальное смещение тени также задается в любой единице измерения, поддерживаемой CSS. Если задано положительное смещение, тень будет расположена ниже текста, если отрицательное — выше. Можно также задать нулевое смещение — тогда тень будет располагаться прямо под текстом.

Радиус размытия тени также задается в любой единице измерения, поддерживаемой CSS. Если этот параметр не указан, или его значение равно нулю, тень не будет иметь эффекта размытия.

Значение `none` (установленное по умолчанию) убирает тень у текста.

В следующем стиле мы задали для заголовков первого уровня тень, расположенную правее и ниже текста на 1 мм и имеющую радиус размытия 1 пиксел:

```
h1 { text-shadow: 1mm 1mm 1px black; }
```

Вывод текста

Еще в *главе 2* мы узнали правила, которыми Web-обозреватель руководствуется при выводе текста. В частности, следующие друг за другом пробелы и табуляции он преобразует в единичный пробел, переводы строк также преобразует в пробелы, а также сам выполняет перенос строк, чтобы вместить текст в свое окно по ширине.

Однако мы имеем возможность управлять выводом текста с помощью средств CSS. Для этого предназначены два атрибута стилей, которые будут рассмотрены далее.

Атрибут стиля `white-space` задает правила вывода текста:

```
white-space: normal|pre|nowrap|pre-wrap|pre-line
```

Этот атрибут может иметь пять значений:

- `normal` — последовательности пробелов и табуляций преобразуются в один пробел, переводы строк преобразуются в пробелы. Выполняется перенос строк. Это поведение по умолчанию для всех тегов, кроме `<pre>` (текста фиксированного форматирования);
- `pre` — последовательности пробелов, табуляций и переводы строк выводятся как есть. Перенос строк не выполняется. Это поведение по умолчанию для тега `<pre>`;
- `nowrap` — последовательности пробелов и табуляций преобразуются в один пробел, переводы строк также преобразуются в пробелы. Перенос строк не выполняется;

- `pre-wrap` — последовательности пробелов, табуляций и переводы строк выводятся как есть. Выполняется перенос строк;
- `pre-line` — последовательности пробелов и табуляций преобразуются в один пробел, переводы строк выводятся как есть. Выполняется перенос строк.

Чтобы читателям было проще выбрать нужное значение атрибута стиля `white-space`, автор свел все доступные для него значения в табл. 9.1.

Таблица 9.1. *Значения атрибута стиля `white-space` и результаты их применения*

<table>
<tr><th>Значение</th><th>Переводы строк</th><th>Последовательности пробелов</th><th>Перенос строк</th></tr>
<tr><td>normal</td><td>Преобразуются в пробелы</td><td>Преобразуются в один пробел</td><td>Выполняется</td></tr>
<tr><td>pre</td><td>Сохраняются</td><td>Сохраняются</td><td rowspan="2">Не выполняется</td></tr>
<tr><td>nowrap</td><td>Преобразуются в пробелы</td><td>Преобразуются в один пробел</td></tr>
<tr><td>pre-wrap</td><td>Сохраняются</td><td rowspan="2">Сохраняются</td><td rowspan="2">Выполняется</td></tr>
<tr><td>pre-line</td><td>Преобразуются в пробелы</td></tr>
</table>

Вот стиль, переопределяющий поведение тега `<pre>` так, чтобы при необходимости его содержимое переносилось по строкам:

```
pre { white-space: pre-wrap; }
```

Атрибут стиля `word-wrap` применяется нечасто, но в некоторых случаях без него не обойтись. Он позволяет указать места, в которых Web-обозреватель может выполнить перенос строк:

```
word-wrap: normal|break-word
```

Здесь доступны два значения:

- `normal` — указывает Web-обозревателю, что он может выполнять перенос строк только по пробелам. Это поведение по умолчанию;
- `break-word` — разрешает Web-обозревателю выполнять перенос строк также и внутри слов. Это может пригодиться, если текст содержит очень много длинных слов, которые по ширине не помещаются в родительский элемент.

Пример:

```
P ( word-wrap: break-word; }
```

Загружаемые шрифты. Директивы CSS

Мы уже знаем, как задать шрифт для текста. Но проблема в том, что этот шрифт уже должен быть установлен на компьютере посетителя. А если его там нет?

Специально для таких случаев CSS поддерживает использование *загружаемых шрифтов*. Такой шрифт хранится в виде файла в составе сайта и загружается самим Web-обозревателем в случае необходимости.

Рекомендуется использовать загружаемые шрифты формата *WOFF* (Web Open Font Format, открытый формат шрифтов Web) версии 1.0 — он поддерживается всеми современными Web-обозревателями и не вызывает проблем при применении.

> ***На заметку***
>
> Для преобразования шрифтов в формат WOFF можно воспользоваться онлайновым сервисом-перекодировщиком Font2Web, находящимся по интернет-адресу **http://www.font2web.com/**.

Загружаемый шрифт указывается с помощью директивы под названием `@font-face`. (*Директивой* называется особая команда языка CSS, которая не управляет параметрами представления, а выполняет какие-либо подготовительные действия или задает дополнительные условия.) Записывается она в следующем формате:

```
@font-face {
  font-family: <имя шрифта>;
  src: url(<интернет-адрес файла шрифта>);
  font-weight: <"жирность" загружаемого шрифта>;
  font-style: <начертание загружаемого шрифта>;
}
```

Как видим, в составе этой директивы указываются четыре параметра:

- `font-family` — указывает имя шрифта, которое можно будет потом использовать в атрибуте стиля `font-family`, чтобы сослаться на этот шрифт;
- `src` — задает интернет-адрес файла со шрифтом, который в этом случае берется в двойные кавычки;
- `font-weight` — указывает «жирность» шрифта, хранящегося в загружаемом файле. Этот параметр поддерживает те же значения, что и одноименный атрибут стиля;
- `font-style` — указывает начертание шрифта, хранящегося в загружаемом файле. Значения этого параметра — те же значения, что у одноименного атрибута стиля.

Последние два параметра указывают степень «жирности» и начертание шрифта, который хранится в файле, чей интернет-адрес задает параметр `src`. Дело в том, что шрифты различных «жирностей» и начертаний — обычные, полужирные, курсивные и полужирные курсивные — хранятся в отдельных файлах, и эти файлы требуется загружать отдельно.

Если среди загруженных отсутствует шрифт требуемой «жирности» или начертания, Web-обозреватель предпримет попытку преобразовать в таковой один из имеющихся шрифтов. К сожалению, как правило, результат получается не очень высокого качества. Поэтому настоятельно рекомендуется загружать шрифты всех используемых на странице «жирностей» и начертаний.

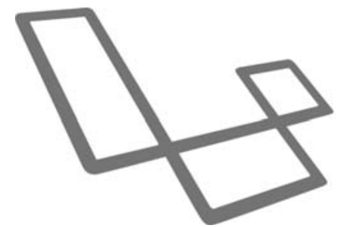
В следующем фрагменте кода мы указываем загрузить два шрифта из файлов `myriadpro.woff` (обычной «жирности») и `myriadprobold.woff` (полужирный), не забыв указать, какой из этих шрифтов обычный, а какой полужирный. Даем для них одинаковое имя `MyriadPro`, под которым эти шрифты будут доступны в стилях:

```
@font-face {
  font-family: MyriadPro;
  src: url("/fonts/myriadpro.woff");
  font-weight: normal;
}
@font-face {
  font-family: MyriadPro;
  src: url("/fonts/myriadprobold.woff");
  font-weight: bold;
}
```

После этого мы можем использовать загруженные шрифты где угодно, указав заданное для них имя:

```
p { font-family: MyriadPro; }
address { font-family: MyriadPro;
          font-weight: bold; }
```

Для вывода абзацев Web-обозреватель использует шрифт из файла `myriadpro.woff`, а для вывода адресов — шрифт из файла `myriadprobold.woff`, поскольку для первых был указан обычный шрифт, а для вторых — полужирный.



ГЛАВА 10

Отображение и видимость элементов. Параметры курсора. Генерируемое содержание

Помимо параметров текста, CSS позволяет нам указывать, как элементы страницы будут отображаться на экране и будут ли они вообще на нем присутствовать (что может пригодиться при создании разметки и интерактивных элементов). Также мы можем задать форму курсора мыши при наведении его на элемент страницы.

А еще мы можем сделать так, чтобы сам Web-обозреватель дополнял содержимое элементов страницы при выводе на экран и даже сам нумеровал их.

Параметры отображения

Начнем мы с атрибутов стилей, управляющих отображением элементов страницы. Один из них указывает, как элемент будет вести себя при выводе на экран: как блочный элемент, как встроенный или как ячейка таблицы, — а другой позволяет сделать его невидимым.

- Атрибут стиля `visibility` как раз и сообщает Web-обозревателю, должен ли элемент отображаться на странице:

  ```
  visibility: visible|hidden|collapse
  ```

  Он может принимать три значения:

  - `visible` — элемент отображается на странице (это обычное поведение);
  - `hidden` — элемент не отображается на странице, однако под него все еще выделено на ней место. Другими словами, вместо элемента на странице видна пустая «прореха»;
  - `collapse` — применим только к таблицам. Строки и ячейки таблицы полностью скрываются, однако под саму таблицу на странице все еще выделяется место. (Говоря по правде, польза от этого весьма сомнительна…)

  Атрибут стиля `visibility` применяется довольно редко и только для создания интерактивных элементов (см. *раздел 3*).

- Атрибут стиля `display` позволяет задать режим отображения для элемента страницы: будет он блочным, встроенным, станет пунктом списка или ячейкой таблицы:

```
display: none|inline|block|inline-block|list-item|run-in|table|
inline-table|table-caption|table-header-group|table-row-group|
table-footer-group|table-row|table-cell
```

  Доступных значений у этого атрибута стиля очень много:

  - `none` — элемент вообще не будет отображаться на странице, словно он и не задан в ее HTML-коде;
  - `inline` — встроенный элемент (значение по умолчанию для встроенных элементов);
  - `block` — блочный элемент (значение по умолчанию для блочных элементов);
  - `inline-block` — блочный элемент, который будет обтекаться содержимым соседних блочных элементов;
  - `list-item` — пункт списка;
  - `run-in` — *встраивающийся* элемент. Если за таким элементом следует блочный элемент, он становится частью этого блочного элемента (фактически — встроенным в него элементом), в противном случае он сам становится блочным элементом;
  - `table` — таблица;
  - `inline-table` — таблица, отформатированная как встроенный элемент. (Оказывается, мы все-таки можем поместить таблицу в абзац! Только кому это нужно?..)
  - `table-caption` — заголовок таблицы;
  - `table-header-group` — секция «шапки» таблицы;
  - `table-row-group` — секция тела таблицы;
  - `table-footer-group` — секция «поддона» таблицы;
  - `table-row` — строка таблицы;
  - `table-cell` — ячейка таблицы;
  - `flex` — блочный элемент с гибкой версткой;
  - `inline-flex` — встроенный элемент с гибкой версткой (с гибкой версткой мы познакомимся в *главе 13*).

  Вот пример стиля, позволяющего графическим изображениям с привязанным стилевым классом `block` отображаться как блочные элементы:

```
img.block { display: block; }
```

  Вот стиль, превращающий пункты списков во встроенные элементы, которые станут выводиться в одну строку:

```
li { display: inline; }
```

А этот стиль делает все элементы с привязанным стилевым классом `hidden` невидимыми:

```
.hidden { display: none; }
```

Более того, на самой странице даже не останется никакого признака того, что эти элементы на ней присутствовали.

В большинстве практических случаев достаточно значений `none`, `block` и `inline` атрибута стиля `display`. Остальные значения слишком специфичны и применяются только в особых случаях, например, при создании разметки (о которой мы поговорим в *главе 13*).

Параметры курсора

Атрибут стиля `cursor` устанавливает форму курсора мыши при наведении его на какой-либо элемент страницы:

```
cursor: auto|default|none|pointer|progress|wait|text|help
```

- `auto` — Web-обозреватель сам управляет формой курсора мыши. Это обычное поведение;
- `default` — курсор по умолчанию, обычно стрелка;
- `none` — курсор мыши вообще не отображается;
- `pointer` — «указующий перст». Обычное поведение при наведении курсора мыши на гиперссылку;
- `progress` — стрелка с небольшими песочными часами. Обозначает, что в данный момент работает какой-то фоновый процесс;
- `wait` — песочные часы. Обозначает, что в данный момент Web-обозреватель занят;
- `text` — текстовый курсор. Обычное поведение при наведении курсора мыши на фрагмент текста;
- `help` — стрелка с вопросительным знаком.

> ***На заметку***
>
> Здесь мы рассмотрели лишь наиболее часто применяемые значения атрибута стиля `cursor`. В реальности он поддерживает гораздо больше значений, которые можно увидеть на странице **http://www.w3schools.com/cssref/pr_class_cursor.asp**.

Вот пример задания курсора мыши в виде «указующего перста» для пунктов списка, вложенного в тег семантической панели навигации (`<nav>`):

```
nav li { cursor: pointer; }
```

Изменение формы курсора мыши часто применяется при создании интерактивных элементов страницы.

Генерируемое содержание

Генерируемое содержание — это содержание, которое не задается в HTML-коде, а формируется самим Web-обозревателем согласно определенным правилам. В качестве примера генерируемого содержания можно привести маркеры и нумерацию пунктов списка, поскольку они вставляются в текст Web-обозревателем, а не пишутся нами.

Стандарт CSS позволяет нам создавать на страницах собственное генерируемое содержание. Это могут быть как простые надписи, вставляемые в текст элементов, так и нумерация, которую можно создать для элементов любого типа, например заголовков.

Статичное генерируемое содержание

Проще всего вывести статичное генерируемое содержание: какую-либо надпись, значение заданного атрибута тега, графическое изображение или кавычки. Например, мы можем вывести надпись «Примечание» перед текстом определенных абзацев.

Для указания статичного генерируемого содержания применяется атрибут стиля `content`. Формат его написания таков:

```
content: normal|none|<текст надписи>|open-quote|close-quote|
no-open-quote|no-close-quote|attr(<имя атрибута тега>)|
url(<интернет-адрес графического файла>)
```

`Текст надписи` и *`интернет-адрес графического файла`* берутся в двойные кавычки. *`Имя атрибута тега`* в кавычки не берется.

Также поддерживаются следующие предопределенные значения:

- `open-quote` — открывающая кавычка;
- `close-quote` — закрывающая кавычка;
- `no-open-quote` — убирает открывающую кавычку, если она была задана унаследованным стилем (о наследовании стилей говорилось в *главе 7*);
- `no-close-quote` — убирает закрывающую кавычку, если она была задана унаследованным стилем;
- `normal` — генерируемое содержание по умолчанию; для пунктов списков это маркер или нумерация, а для прочих элементов — отсутствие какого-либо содержания (поведение по умолчанию);
- `none` — отсутствие генерируемого содержания.

Селектор стиля, в котором находится этот атрибут, должен указывать на элемент, для которого назначается генерируемое содержание.

Но указать само генерируемое содержание — это еще полдела. Нам также следует задать его местоположение относительно содержимого тега, в котором оно будет присутствовать. Для этого применяются два особых псевдокласса:

- `:before` — вывести генерируемое содержание перед содержимым тега;
- `:after` — вывести генерируемое содержание после содержимого тега.

Эти псевдоклассы указываются также в селекторе стиля, в котором присутствует атрибут `content`.

Примеры:

- `p.note:before { content: "Примечание: "; }`

  Здесь мы задаем для абзацев с привязанным стилевым классом `note` в качестве генерируемого содержания текст `"Примечание"`, двоеточие и пробел — это содержание будет добавлено в начало текста абзацев.
- `img:after { content: attr(alt); }`

  Добавляем к изображениям заданный для них текст замены (значение атрибута тега `alt`).
- `h1:before { content: url("/images/dot.gif"); }`

  Выводим перед текстом заголовков первого уровня графическое изображение.
- А здесь мы берем текст заголовков первого уровня в кавычки:

  ```
  h1:before { content: open-quote; }
  h1:after { content: close-quote; }
  ```

Наконец, мы можем объединять указанные ранее варианты значений атрибута стиля `content`, разделяя их пробелами, чтобы создать более сложное генерируемое содержание:

```
p.note:before { content: "Примечание: " open-quote; }
p.note:after { content: close-quote; }
```

В результате, в начале всех абзацев с привязанным стилевым классом `note` будет присутствовать текст «Примечание», двоеточие, пробел и открывающая кавычка, а в их конце будет добавлена закрывающая кавычка.

Создание нумерации

Нумерация создается несколько сложнее, чем статичное генерируемое содержание. Сначала нам потребуется создать счетчик, а затем указать его значение в качестве генерируемого содержания для нужных элементов страницы.

Счетчик CSS можно рассматривать как ячейку памяти, хранящую число, которое будет увеличиваться на единицу (или другое заданное нами значение) каждый раз, когда в HTML-коде встретится заданный нами элемент страницы. В результате, при достижении каждого такого элемента страницы в счетчике окажется его порядковый номер, который мы можем вывести на экран, создав тем самым нумерацию.

Счетчик создается с помощью атрибута тега `counter-increment`:

```
counter-increment: none | <имя счетчика> [<значение инкремента счетчика>]
```

Имя счетчика указывается без кавычек и должно быть уникальным в пределах страницы. Имена счетчиков должны включать только латинские буквы, цифры и знаки

дефиса, причем начинаться они должны с буквы, пробелы в именах счетчиков не допускаются.

Необязательное *значение инкремента счетчика* указывает, на сколько будет увеличиваться значение счетчика каждый раз, когда в HTML-коде встретится элемент страницы указанного нами типа. Другими словами, это величина, на которую будут различаться номера двух соседних пронумерованных элементов страницы. Это значение должно быть целым числом, допускаются как положительные, так и отрицательные значения. Если оно не указано, значение счетчика будет каждый раз увеличиваться на 1.

Значение `none` указывает, что никакие счетчики не должны инкрементироваться. Это значение по умолчанию.

Сразу при создании счетчик получит в качестве начального значения 0. Встретив подходящий к селектору элемент страницы, Web-обозреватель сначала увеличит значение счетчика, а уже потом выведет его на экран. Не забываем это.

Чтобы создать счетчик, следует написать стиль, селектор которого указывает на элементы, что нам требуется пронумеровать, и вставить в этот стиль атрибут `counter-increment`. Например, с помощью такого стиля мы можем нумеровать заголовки первого уровня:

```
h1 { counter-increment: header1; }
```

Для вывода значения счетчика применяется уже знакомый нам атрибут стиля `content`. Только формат записи его значения будет таким:

```
counter(<имя счетчика>[, <тип нумерации>])
```

Тип нумерации задается в том же формате, что тип нумерации для пунктов списка (см. *главу 9*, описание атрибута стиля `list-style-type`).

Пара примеров:

- создаем у заголовков первого уровня нумерацию вида 1, 2, 3... Не забываем отделить ее от содержимого элемента пробелом:

  ```
  h1 { counter-increment: header1; }
  h1:before { content: counter(header1) " "; }
  ```

- А здесь мы создаем аналогичную нумерацию, выполненную римскими цифрами и предваренную словом «Часть» и пробелом:

  ```
  h1 { counter-increment: header1; }
  h1:before { content: "Часть " counter(header1, upper-latin) " "; }
  ```

Атрибут стиля `counter-reset` указывает при достижении элемента страницы заданного типа занести в счетчик значение 0, тем самым позволяя начать нумерацию заново. Другими словами, этот атрибут стиля выполняет *сброс счетчика*:

```
counter-reset: none | <имя счетчика> [<начальное значение счетчика>]
```

Понятно, что будет выполнен сброс счетчика с указанным *именем*. *Начальное значение счетчика* указывает значение, которое счетчик получит в результате сброса. Если мы его не укажем, счетчик получит значение 0.

Значение `none` указывает, что никакие счетчики не должны сбрасываться. Это значение по умолчанию.

Атрибут `counter-reset` указывается в стиле, чей селектор должен совпадать с элементами, на которых требуется выполнять сброс счетчика.

Пример:

```
h1 { counter-reset: header2; }
```

Теперь на каждом заголовке первого уровня счетчик `header2` будет сбрасываться.

Набор стилей, приведенный в листинге 10.1, создаст многоуровневую нумерацию следующего вида:

```
1. Глава
  1.1. Параграф
  1.2. Параграф
2. Глава
  2.1. Параграф
  2.2. Параграф
```

Листинг 10.1

```
h1 { counter-increment: header1;
     counter-reset header2; }
h2 { counter-increment: header2; }
h1:before {content: counter(header1) ". "; }
h2:before {content: counter(header1) "." counter(header2) ". "; }
```

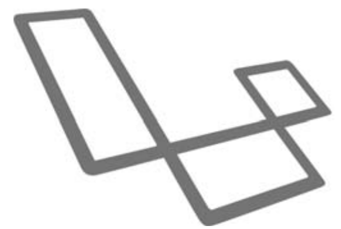
Осталось заметить, что мы можем указать в значении атрибута стиля `counter-increment` сразу несколько счетчиков, разделив их определения пробелами:

```
h1 { counter-increment: header1 header2; }
h2 { counter-increment: header1 header2 2; }
```

Второй стиль указывает для счетчика `header2` величину инкремента 2.

В атрибуте стиля `counter-reset` также можно указать несколько счетчиков:

```
h1 { counter-reset: header2 header3; }
```



ГЛАВА 11

Параметры фона. Градиентные фоны CSS 3

Средства по указанию фона элементов страницы, поддерживаемые CSS, исключительно богаты. Мы можем задать в качестве фона обычный цвет, графическое изображение и даже градиент. Также есть возможность задать размеры графического изображения, используемого в качестве фона, его местоположение и установить часть элемента, которая должна быть заполнена этим фоном.

Сплошной фон

Проще всего указать в качестве фона обычный сплошной цвет, создав тем самым *сплошной фон*, — благо базовые средства для этого появились еще в первой версии стандарта CSS.

Атрибут стиля `background-color` служит для задания цвета сплошного фона:

```
background-color: transparent|<цвет>
```

Значение `transparent` убирает фон совсем, при этом элемент страницы получит «прозрачный» фон. По умолчанию фон у элементов страницы отсутствует, а фон самой страницы задает Web-обозреватель, и это фон белого цвета.

Так мы, например, можем задать для всей страницы черный фон и белый текст:

```
body { color: white;
       background-color: black; }
```

Атрибут стиля `background-clip` указывает Web-обозревателю, какая часть элемента страницы должна быть заполнена фоном (или, говоря другими словами, режим заполнения):

```
background-clip: content-box|padding-box|border-box
```

Здесь поддерживаются три предопределенные значения:

- `border-box` — фоном будет заполнено содержимое элемента и пространство, занятое его внутренними отступами и рамкой (о том, как задать внутренние отступы и рамку, будет идти речь в *главе 12*). Это поведение по умолчанию;

- `padding-box` — фоном будет заполнено содержимое элемента и пространство, занятое его внутренними отступами;
- `content-box` — фоном будет заполнено только собственно содержимое элемента.

Так мы заполняем черным фоном лишь содержимое страницы — получится некрасиво, но это лишь пример:

```
body { color: white;
       background-color: black;
       background-clip: content-box; }
```

Графический фон

Задать в качестве фона графическое изображение (*фоновое изображение*), создав тем самым *графический фон*, также несложно.

Создание графического фона

Атрибут стиля `background-image` указывает само фоновое изображение:

```
background-image: url(<интернет-адрес файла изображения>)|none
```

`Интернет-адрес файла фонового изображения` заключается в кавычки.

Значение `none` убирает фоновое изображение.

Пример:

```
table.bgr { background-image: url("/table_background.png"); }
```

Фоновое изображение выводится поверх сплошного фона, заданного нами с помощью атрибута стиля `background-color`. И, если фоновое изображение содержит «прозрачные» фрагменты (такую возможность поддерживают форматы GIF, PNG и SVG), этот фон будет «просвечивать» сквозь них.

В следующем примере мы задали для таблицы и обычный, и графический фон. Это, кстати, распространенная практика в Web-верстке:

```
table.yellow { background-color: yellow;
               background-image: url("/yellow_background.png"); }
```

Одно из примечательных нововведений CSS 3 — возможность указать для элемента страницы сразу нескольких фоновых изображений, перечислив их через запятую. В этом случае изображения будут накладываться друг на друга в порядке, обратном тому, в котором они перечислены в атрибуте стиля `background-image`.

Зададим для таблицы сразу два фоновых изображения:

```
table.yellow { background-image: url("/images/bg1.png"),
               url("/images/bg2.png"); }
```

При этом изображение из файла `bg2.png` будет перекрыто изображением из файла `bg1.png`.

Параметры графического фона

Мы можем указать для графического фона некоторые параметры: размер, местоположение, его повторение и т. п.

Если фоновое изображение меньше, чем элемент страницы (или сама страница), для которого оно задано, Web-обозреватель будет повторять это изображение, пока не «замостит» им весь элемент. Параметры этого повторения задает атрибут стиля `background-repeat`:

```
background-repeat: no-repeat|repeat|repeat-x|repeat-y
```

Здесь доступны четыре значения:

- `repeat` — фоновое изображение будет повторяться по горизонтали и вертикали (обычное поведение);
- `repeat-x` — фоновое изображение будет повторяться только по горизонтали;
- `repeat-y` — фоновое изображение будет повторяться только по вертикали;
- `no-repeat` — фоновое изображение не будет повторяться никогда. В этом случае часть фона элемента страницы, возможно, останется не заполненной им.

В следующем примере заданное нами для страницы фоновое изображение не будет повторяться:

```
body { background-image: url("/images/background.jpg");
       background-repeat: no-repeat; }
```

Остальные параметры графического фона имеет смысл указывать только в том случае, если фоновое изображение не повторяется.

Атрибут стиля `background-size` обозначает размеры фонового изображения:

```
background-size: <ширина> [<высота>] | auto | cover | contain
```

Числовые значения размеров мы можем указать в любой единице измерения, поддерживаемой CSS. Если указаны и ширина, и высота, изображение примет эти размеры. Если указана только ширина, высота будет установлена самим Web-обозревателем такой, чтобы не нарушить пропорции изображения.

Рассмотрим несколько примеров:

- задаем для фонового изображения размеры 640×480 пикселов:

  ```
  body { background-size: 640px 480px; }
  ```

- растягиваем изображение на всю ширину страницы и на половину ее высоты:

  ```
  body { background-size: 100% 50%; }
  ```

- задаем для фонового изображения ширину в 800 пикселов. Высоту установит сам Web-обозреватель:

  ```
  body { background-size: 800px; }
  ```

Также поддерживаются следующие предопределенные значения:

- `auto` — не изменяет размеры фонового изображения (поведение по умолчанию);
- `cover` — задает такие размеры фонового изображения, чтобы оно полностью покрывало элемент страницы, при этом какие-то фрагменты изображения могут выйти за границы элемента;
- `contain` — задает такие размеры фонового изображения, чтобы оно полностью помещалось в элементе страницы, при этом какие-то части элемента могут оказаться не покрытыми графическим фоном.

В следующем примере мы заполняем фоновым изображением все пространство страницы:

```
body { background-size: cover; }
```

Атрибут стиля `background-position` указывает местоположение фонового изображения относительно элемента страницы:

```
background-position: <горизонтальная позиция> [<вертикальная позиция>]
```

Горизонтальная позиция фонового изображения задается в следующем формате:

```
<числовое значение>|left|center|right
```

Помимо *числового значения* расстояния от левой границы элемента до левой границы графического фона, мы можем указать следующие предопределенные значения:

- `left` — фоновое изображение прижимается к левому краю элемента страницы (это обычное поведение);
- `center` — располагается по центру;
- `right` — прижимается к правому краю.

Формат задания *вертикальной позиции* фонового изображения таков:

```
<числовое значение>|top|center|bottom
```

Числовое значение задает расстояние от верхней границы элемента до верхней границы фонового изображения. Еще поддерживаются следующие предопределенные значения:

- `top` — фоновое изображение прижимается к верхнему краю элемента страницы (это обычное поведение);
- `center` — располагается по центру;
- `bottom` — прижимается к нижнему краю.

Если для какого-либо элемента страницы указана только позиция фонового изображения по горизонтали, его вертикальная позиция принимается равной `center`.

Стиль, приведенный в следующем примере, помещает фоновое изображение в центре таблицы и прижимает его к верхнему краю этого элемента:

```
table.bgr ( background-position: center top; }
```

Атрибут стиля `background-origin` позволяет указать, относительно чего будет позиционироваться фоновое изображение (режим позиционирования):

```
background-origin: padding-box|border-box|content-box
```

Доступны три предопределенных значения:

- `padding-box` — изображение будет позиционироваться относительно границ внутренних отступов элемента страницы (поведение по умолчанию);
- `border-box` — позиционирование фонового изображения будет выполняться относительно рамки элемента;
- `content-box` — изображение будет позиционироваться относительно границ содержимого элемента.

В следующем примере мы позиционируем фоновое изображение относительно содержимого таблицы:

```
table.bgr ( background-position: center top;
            background-origin: content-box;
            background-repeat: no-repeat; }
```

В случае графического фона также поддерживается атрибут стиля `background-clip`, указывающий часть элемента, которая должна быть покрыта фоном.

Когда мы прокручиваем содержимое страницы в окне Web-обозревателя, вместе с ней прокручивается и фоновое изображение. CSS предлагает забавную возможность — запрет прокрутки графического фона страницы и фиксация его на месте. Фиксацией фона управляет атрибут стиля `background-attachment`:

```
background-attachment: scroll|fixed|local
```

Значение `scroll` заставляет Web-обозреватель прокручивать фон вместе с содержимым страницы (это поведение по умолчанию). Значение `fixed` фиксирует фон на месте, и он прокручиваться не будет.

Значение `local` применимо лишь для элементов страницы с возможностью прокрутки содержимого. (Как создавать такие элементы, мы узнаем в *главе 12*.) Если оно указано, фоном будет покрыт не сам этот элемент, а его содержимое, — соответственно, при прокрутке содержимого элемента будет прокручиваться и фон.

Пример:

```
div#main { background-image: url(/images/main_background.png);
           background-attachment: fixed; }
```

Ранее мы выяснили, что имеется возможность указать для элемента страницы сразу несколько фоновых изображений, перечислив их через запятую. При этом изображения будут выводиться друг поверх друга в порядке, обратном тому, в котором они перечислены. Мы можем указать для каждого из этих изображений отдельные параметры — также перечислив их через запятую.

В следующем примере для первого фонового изображения мы включаем повторение, а для второго — наоборот, отключаем:

```
table.yellow { background-image: url("/images/bg1.png"),
               url("/images/bg2.png");
               background-repeat: repeat, no-repeat; }
```

Задание сразу всех параметров фона

Осталось рассмотреть «всеобъемлющий» атрибут стиля `background`, который позволяет указать все параметры фона одновременно:

```
background: <цвет фона> <фоновое изображение> <расположение>/<размеры>
<повторение> <позиционирование> <заполнение> <фиксация>
```

Пример:

```
body { background: #ffffff url("/images/background.png") center/cover
       no-repeat content-box content-box fixed; }
```

Градиентный фон

И, наконец, теперь, с появлением CSS 3, мы можем создать у элемента страницы *градиентный фон*.

Введение в градиенты и градиентные фоны

Градиент — это изображение, полученное в результате плавного перетекания одного цвета в другой. В качестве примера градиента можно привести заливку заголовков окон в стандартной теме Windows XP, Vista и 7 — там синий цвет плавно перетекает в светло-голубой.

Градиенты бывают двух видов:

- в *линейном градиенте* переходящие друг в друга цвета распространяются по прямой из одной точки в другую. Точка, в которой градиент начинается, называется *начальной*, а точка, в которой он заканчивается, — *конечной*;
- в *радиальном градиенте* переходящие друг в друга цвета распространяются из начальной точки во все стороны, в конечном счете образуя своего рода эллипс (или, в его «вырожденном» случае, круг). За конечную точку такого градиента принимается любая точка, расположенная на воображаемом эллипсе, охватывающем градиент.

Первое, что нам нужно сделать для создания градиента, — указать его основные параметры: направление — для линейного градиента, форму, местоположение центра охватывающего его эллипса и размеры последнего — для радиального.

После этого мы должны создать набор ключевых точек. *Ключевая точка* располагается на воображаемой прямой, проведенной между начальной и конечной точками градиента, и определяет, какой «чистый» цвет должен присутствовать в этом месте. (При этом цвета, которые будут присутствовать между ключевыми точками,

сформируются путем смешивания «чистых» цветов, заданных в соседних ключевых точках.) Ключевая точка характеризуется расстоянием между ней и начальной точкой градиента и, собственно, значением цвета.

Любой градиент должен иметь, по крайней мере, две ключевые точки, которые часто располагаются, соответственно, в его начальной и конечной точках. Более сложные градиенты имеют более двух ключевых точек, при этом цвет, указанный в первой ключевой точке, будет плавно перетекать в тот, что указан во второй ключевой точке, потом — в цвет из третьей ключевой точки и т. д.

Градиентный фон указывается либо с помощью атрибута стиля `background-image`, либо в составе атрибута стиля `background`.

Создание линейного градиента

Линейные градиенты создаются проще всего. Их описание формируется с применением функции `linear-gradient` в формате, показанном в листинге 11.1.

Листинг 11.1

```
linear-gradient(
  [to <направление> | <угол> ,]
  <цвет в ключевой точке 1>[ <местоположение ключевой точки 1>],
  <цвет в ключевой точке 2>[ <местоположение ключевой точки 2>],
  . . .
  <цвет в ключевой точке n-1>[ <местоположение ключевой точки n-1>],
  <цвет в ключевой точке n>[ <местоположение ключевой точки n>]
)
```

Направление, по которому будет распространяться градиент, можно указать двумя способами:

- первый способ предусматривает применение следующих предопределенных значений: `top` (вверх), `bottom` (вниз), `left` (налево) и `right` (направо). В этом случае градиент станет распространяться в указанном направлении по прямой линии — горизонтальной или вертикальной, конечная точка будет находиться на той стороне элемента страницы, на которой заканчивается эта линия, а начальная — на его противоположной стороне.

  Также мы можем указать комбинацию из двух таких направлений, разделенных пробелом. В этом случае градиент будет распространяться по диагонали — в угол элемента, образованный смыканием указанных сторон. Конечная точка градиента поместится в углу, образованном смыкающимися сторонами, а начальная точка — в противоположном углу;
- второй способ предполагает указание направления градиента — задание его в виде угла между горизонталью и линией, по которой распространяется градиент, отсчитывается этот угол по часовой стрелке. Если требуется отсчитывать угол против часовой стрелки, нужно указать отрицательное значение.

Если направление не указано, градиент будет распространяться сверху вниз (как будто для него задано значение `bottom`).

Местоположение ключевой точки — расстояние от начальной точки градиента до нее — обычно указывают в процентах. Тогда градиент примет точно такие же размеры, что и сам элемент.

Если местоположение первой ключевой точки градиента не указано, она будет находиться на начальной точке. Если не указать местоположение последней ключевой точки, она будет находиться на конечной точке. А если не указывать местоположения промежуточных ключевых точек, они выстроятся по градиенту на одинаковом расстоянии друг от друга.

Рассмотрим несколько примеров:

- красно-синий градиент, распространяющийся по вертикали сверху вниз (поскольку направление не задано явно). Местоположение ключевых точек также не указано, следовательно, первая будет находиться на начальной точке градиента, а вторая — на конечной:

  ```
  background-image: linear-gradient(red, blue);
  ```

- такой же градиент, но распространяющийся по горизонтали слева направо:

  ```
  background-image: linear-gradient(to right, red, blue);
  ```

- градиент, состоящий из трех цветов. Третья ключевая точка, задающая синий цвет, поместится на середине (50%) элемента, оставшаяся часть элемента будет закрашена чистым синим цветом. Вторая ключевая точка, задающая черный цвет, поместится между первой и третьей ключевыми точками (на 25% от ширины элемента):

  ```
  background-image: linear-gradient(to right, red, black, blue 50%);
  ```

- такой же градиент, распространяющийся из левого нижнего в правый верхний угол элемента:

  ```
  background-image: linear-gradient(to right top, red, black, blue 50%);
  ```

- диагональный градиент, распространяющийся по прямой, которая наклонена под углом 45° к горизонтали, если считать в направлении часовой стрелки:

  ```
  background-image: linear-gradient(45deg, red, black, blue 50%);
  ```

- градиент, включающий пять цветов:

  ```
  background-image: linear-gradient(red, black 10%, green 40%,
  white 85%, blue);
  ```

Создание радиального градиента

Описание радиального градиента включает функцию `radial-gradient` и выглядит несколько сложнее. Его формат представлен в листинге 11.2.

Листинг 11.2

```
radial-gradient(
  [[<форма>] [<размер>] [at <местоположение начальной точки>],]
  <цвет в ключевой точке 1>[ <местоположение ключевой точки 1>],
  <цвет в ключевой точке 2>[ <местоположение ключевой точки 2>],
  . . .
  <цвет в ключевой точке n-1>[ <местоположение ключевой точки n-1>],
  <цвет в ключевой точке n>[ <местоположение ключевой точки n>]
)
```

Для параметра *форма* доступны значения `ellipse` (градиент эллиптической формы — это форма по умолчанию) и `circle` (градиент в форме круга).

Для эллиптического градиента эллипс, на основе которого он формируется, будет иметь то же соотношение сторон, что и элемент, фоном которого он станет.

Размер градиента можно указать в виде одного из указанных далее значений:

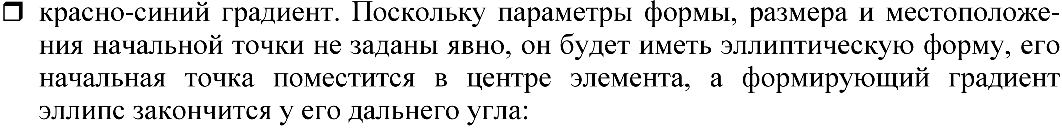
- `closest-side` — градиент будет заканчиваться у самых близких к его начальной точке сторон элемента страницы;
- `closest-corner` — градиент будет заканчиваться у самого близкого к его начальной точке угла элемента страницы;
- `farthest-side` — градиент будет заканчиваться у самых дальних от его начальной точки сторон элемента страницы;
- `farthest-corner` — градиент будет заканчиваться у самого дальнего от его начальной точки угла элемента страницы (значение по умолчанию).

Размер градиента также можно указать в виде одного или двух числовых значений, разделенных пробелом. Если указано одно значение, оно задаст и ширину, и высоту градиента. Если указаны два значения, первое задаст ширину градиента, а второе — его высоту.

Местоположение начальной точки радиального градиента задается в том же формате, что и значение атрибута стиля `background-position`. Если местоположение не указано, начальная точка будет располагаться в центре элемента.

А набор ключевых точек задается точно таким же образом, как и у линейного градиента.

Вот несколько примеров:



- красно-синий градиент. Поскольку параметры формы, размера и местоположения начальной точки не заданы явно, он будет иметь эллиптическую форму, его начальная точка поместится в центре элемента, а формирующий градиент эллипс закончится у его дальнего угла:

  ```
  background-image: radial-gradient(red, blue);
  ```

- бело-синий градиент также эллиптической формы, чья начальная точка находится в правом нижнем углу элемента, а формирующий градиент эллипс заканчивается у его самых дальних сторон:

```
background-image: radial-gradient(farthest-side at right bottom,
white, blue);
```

- аналогичный градиент, но круглой формы:

```
background-image: radial-gradient(circle farthest-side at right
bottom, white, blue);
```

- сложный градиент с несколькими ключевыми точками:

```
background-image: radial-gradient(circle farthest-side at right
bottom, white, red 10%, black 45%, blue 60%);
```

Создание повторяющегося градиента

Нам совсем необязательно устанавливать последнюю ключевую точку на месте конечной точки градиента. Мы можем установить ее на другое место, скажем, посередине элемента, задав для нее соответствующее местоположение:

```
background-image: linear-gradient(red, blue 50%);
```

При этом оставшаяся часть элемента, не «охваченная» градиентом, будет закрашена тем цветом, что указан в последней ключевой точке. В нашем случае оставшаяся половина элемента будет залита «чистым» синим цветом.

Но CSS 3 позволяет нам создать *повторяющийся градиент*. Такой градиент будет повторяться за границей, обозначенной последней ключевой точкой, пока не заполнит элемент страницы целиком.

Для создания повторяющихся градиентов используются функции `repeating-linear-gradient` и `repeating-radial-gradient`:

```
repeating-linear-gradient(<параметры линейного градиента>)
repeating-radial-gradient(<параметры радиального градиента>)
```

Первая функция создает линейный градиент, вторая — радиальный. Они принимают те же самые аргументы, что и уже знакомые нам функции `linear-gradient` и `radial-gradient`.

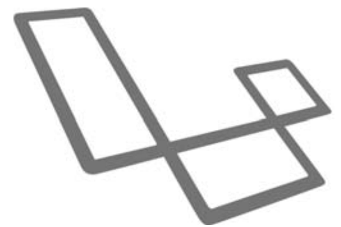
Вот пара примеров:

- повторяющийся линейный красно-синий градиент:

```
background-image: repeating-linear-gradient(red, blue 50%);
```

- повторяющийся радиальный градиент с несколькими ключевыми точками:

```
background-image: repeating-radial-gradient(circle farthest-side at
                  right bottom, white, red 10%, black 45%, blue 60%);
```



ГЛАВА 12

Размеры, отступы, рамки, тени и выделение. Параметры таблиц

Целый набор атрибутов стилей CSS предназначен для указания отступов, позволяющих отодвинуть содержимое элемента от его границы или отодвинуть сам элемент от границ родителя и соседних элементов. Другие атрибуты стилей задают для элементов параметры рамки, рисуемой по их границе, размеры, режим управления переполнением, выделение и тень, отбрасываемую на этот раз не текстом, а самим элементом.

А еще существуют особые атрибуты стилей, которые устанавливают параметры, специфичные для таблиц. Их совсем немного, но во многих случаях они очень нам пригодятся.

Скажем сразу, что почти все эти параметры мы можем указать лишь для блочных элементов. Исключением является лишь выделение — мы можем создать его как у блочных, так и у встроенных элементов страницы.

Параметры отступов

Стандарт CSS предлагает средства для создания отступов двух видов:

- отступ между воображаемой границей элемента страницы и его содержимым — *внутренний* отступ. Внутренний отступ позволяет отодвинуть содержимое элемента от его границы;
- отступ между воображаемой границей того или иного элемента страницы и воображаемыми границами его родителя и соседних элементов — *внешний* отступ. Внешний отступ создает просвет между элементом и границами его родителя и соседних элементов.

Атрибуты стилей `padding-left`, `padding-top`, `padding-right` и `padding-bottom` служат для указания величин внутренних отступов, соответственно, слева, сверху, справа и снизу:

```
padding-left|padding-top|padding-right|padding-bottom: <отступ>
```

Мы можем указать в качестве величины отступа абсолютное или относительное значение. По умолчанию величина отступа, как правило, равна нулю.

В листинге 12.1 мы указали для ячеек таблиц отступ, равный двум пикселам со всех сторон.

Листинг 12.1

```
th, th { padding-left: 2px;
         padding-top: 2px;
         padding-right: 2px;
         padding-bottom: 2px; }
```

А вот стиль, создающий внутренние отступы, равные двум сантиметрам слева и справа:

```
.indented { padding-left: 2cm;
            padding-right: 2cm; }
```

Он будет действовать на все блочные элементы страницы, к которым мы привяжем стилевой класс `indented`.

Атрибут стиля `padding` указывает величины внутренних отступов сразу со всех сторон элемента страницы:

```
padding: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]
```

- Если указаны четыре значения, первое задаст величину отступа сверху, второе — справа, третье — снизу, а четвертое — слева.
- Если указаны три значения, первое определит величину отступа сверху, второе — слева и справа, а третье — снизу.
- Если указаны два значения, первое установит величину отступа сверху и снизу, а второе — слева и справа.
- Если указано одно значение, оно задаст величину отступа со всех сторон элемента страницы.

В следующих примерах мы просто переписали определения приведенных ранее стилей с использованием атрибута стиля `padding`:

```
td, th { padding: 2px; }
.indented { padding: 0cm 2cm 0cm 2cm; }
```

Атрибуты стилей `margin-left`, `margin-top`, `margin-right` и `margin-bottom` задают величины внешних отступов, соответственно, слева, сверху, справа и снизу:

```
margin-left|margin-top|margin-right|margin-bottom: <отступ>|auto
```

Значение отступа по умолчанию зависит, во-первых, от Web-обозревателя, а во-вторых, от типа элемента страницы. Так, у блока отступ будет равен нулю, а у абзаца окажется ненулевым.

Этот стиль, например, создаст у всех заголовков первого уровня отступ сверху в 5 мм:

```
h1 { margin-top: 5mm; }
```

В качестве значений внешних отступов допустимы отрицательные величины:

```
li { margin-left: -20px; }
```

В этом случае Web-обозреватель создаст выступ. Такой прием позволяет убрать отступы, создаваемые Web-обозревателем по умолчанию, — например, отступы слева у больших цитат и пунктов списков.

Все современные Web-обозреватели позволяют выполнить один трюк. Если мы зададим для элемента страницы ширину (как это сделать, будет рассмотрено далее) и укажем ему для внешних отступов слева и справа значение `auto`, то такой элемент (здесь — блок `container`) будет расположен по центру страницы:

```
div#container { width: 960px;
                margin-left: auto;
                margin-right: auto; }
```

Внешние отступы мы также можем указать с помощью атрибута стиля `margin`. Он задает величины отступа одновременно со всех сторон элемента страницы:

```
margin: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]
```

Этот атрибут стиля ведет себя так же, как его «коллега» `padding`:

```
h1 { margin: 5mm 0mm; }
div#container { width: 960px;
                margin 0px auto; }
```

Параметры рамки

CSS позволяет нам создать сплошную, пунктирную или точечную рамку, которая будет выводиться по воображаемой границе элемента.

Атрибуты стилей `border-left-style`, `border-top-style`, `border-right-style` и `border-bottom-style` задают стиль линий, которыми будет нарисована, соответственно, левая, верхняя, правая и нижняя сторона рамки:

```
border-left-style|border-top-style|border-right-style|
border-bottom-style: none|hidden|dotted|dashed|solid|double|groove|
ridge|inset|outset
```

Здесь доступны следующие значения:

- `none` и `hidden` — рамка отсутствует (поведение по умолчанию);
- `dotted` — пунктирная линия;
- `dashed` — штриховая линия;
- `solid` — сплошная линия;
- `double` — двойная линия;
- `groove` — «вдавленная» трехмерная линия;
- `ridge` — «выпуклая» трехмерная линия;

- `inset` — трехмерная «выпуклость»;
- `outset` — трехмерное «углубление».

Листинг 12.2 представляет стиль, задающий для ячеек таблиц пунктирную рамку со всех сторон.

Листинг 12.2

```
td, th { border-left-style: dotted;
         border-top-style: dotted;
         border-right-style: dotted;
         border-bottom-style: dotted; }
```

Еще пример:

```
h1 { border-bottom-style: double; }
```

Здесь у заголовков первого уровня мы задаем только стиль нижней стороны рамки, в результате чего они будут подчеркнуты двойной линией.

Атрибут стиля `border-style` позволяет указать стиль сразу для всех сторон рамки:

```
border-style: <стиль 1> [<стиль 2> [<стиль 3> [<стиль 4>]]]
```

- Если указаны четыре значения, первое задаст стиль верхней, второе — правой, третье — нижней, а четвертое — левой стороны рамки.
- Если указаны три значения, первое задаст стиль верхней, второе — левой и правой, а третье — нижней стороны рамки.
- Если указаны два значения, первое задаст стиль верхней и нижней, а второе — левой и правой сторон рамки.
- Если указано одно значение, оно задаст стиль всех сторон рамки.

Пример:

```
td, th { border-style: dotted; }
```

Атрибуты стилей `border-left-width`, `border-top-width`, `border-right-width` и `border-bottom-width` задают толщину, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-width|border-top-width|border-right-width|
border-bottom-width: thin|medium|thick|<толщина>
```

Мы можем указать либо числовое значение толщины рамки, либо одно из предопределенных значений: `thin` (тонкая), `medium` (средняя) или `thick` (толстая). В последнем случае реальная толщина рамки зависит от Web-обозревателя. Значение толщины по умолчанию — `medium`.

В листинге 12.3 мы создали у ячеек таблиц тонкую пунктирную рамку со всех сторон.

Листинг 12.3

```
td, th { border-style: dotted;
         border-left-width: thin;
         border-top-width: thin;
         border-right-width: thin;
         border-bottom-width: thin; }
```

А вот стиль, задающий толщину нижней стороны рамки равной 5 пикселам:

```
h1 { border-bottom-style: double;
     border-bottom-width: 5px; }
```

Атрибут стиля `border-width` указывает значения толщины сразу для всех сторон рамки:

```
border-width: <толщина 1> [<толщина 2> [<толщина 3> [<толщина 4>]]]
```

Он ведет себя так же, как и знакомый нам атрибут стиля `border-style`.

Пример:

```
td, th { border-style: dotted;
         border-width: thin; }
```

Атрибуты стилей `border-left-color`, `border-top-color`, `border-right-color` и `border-bottom-color` задают цвет, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-color|border-top-color|border-right-color|
border-bottom-color: transparent|<цвет>
```

Значение `transparent` задает «прозрачный» цвет, сквозь который будет «просвечивать» фон родительского элемента. Если цвет не указан, рамка будет выведена цветом, заданным для текста элемента.

Пример:

```
h1 { bordet-bottom-style: double;
     border-bottom-width: 5px;
     border-bottom-color: red; }
```

Атрибут стиля `border-color` позволяет указать цвет сразу для всех сторон рамки:

```
border-color: <цвет 1> [<цвет 2> [<цвет 3> [<цвет 4>]]]
```

Он ведет себя так же, как и аналогичный атрибут стиля `border-style`.

Пример:

```
td, th { border-style: dotted;
         border-width: thin;
         border-color: black; }
```

Атрибуты стилей `border-left`, `border-top`, `border-right` и `border-bottom` задают сразу все параметры для, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left|border-top|border-right|border-bottom:
<толщина> <стиль> <цвет>
```

Во многих случаях эти атрибуты стиля оказываются предпочтительнее, поскольку указывают все параметры в одном месте.

Пример:

```
h1 { border-bottom: 5px double red; }
```

«Глобальный» атрибут стиля `border` позволяет установить все параметры сразу для всех сторон рамки:

```
border: <толщина> <стиль> <цвет>
```

Пример:

```
td, th { border: thin dotted black; }
```

Атрибуты стилей `border-top-left-radius`, `border-top-right-radius`, `border-bottom-right-radius` и `border-bottom-left-radius` указывают радиус скругления, соответственно, левого верхнего, правого верхнего, правого нижнего и левого нижнего углов рамки:

```
border-top-left-radius|border-top-right-radius|
border-bottom-right-radius|border-bottom-left-radius:
<радиус горизонтальной четверти> [<радиус вертикальной четверти>]
```

Если *радиус вертикальной четверти* не задан, его значение будет таким же, как у *радиуса горизонтальной четверти*. Значение радиуса скругления по умолчанию — 0 (т. е. угол рамки не скруглен).

Код, представленный в листинге 12.4, задает для всех углов рамки радиус скругления, равный 2-м пикселам.

Листинг 12.4

```
td, th { border-top-left-radius: 2px;
         border-top-right-radius: 2px;
         border-bottom-right-radius: 2px;
         border-bottom-left-radius: 2px; }
```

Атрибут стиля `border-radius` можно использовать для задания радиусов скругления сразу для всех углов рамки:

```
border-radius: <радиус горизонтальной четверти 1>
[<радиус горизонтальной четверти 2> [<радиус горизонтальной четверти 3>
[<радиус горизонтальной четверти 4>]]]
[/<радиус вертикальной четверти 1> [<радиус вертикальной четверти 2>
[<радиус вертикальной четверти 3> [<радиус вертикальной четверти 4>]]]]
```

Он ведет себя так же, как и атрибут стиля `border-style`.

В следующем примере мы задаем для всех углов рамки радиус горизонтальной четверти в 2 пиксела, а радиус вертикальной четверти — в 3 пиксела:

```
td, th { border-radius: 2px / 3px; }
```

Параметры размеров

Атрибуты стилей `width` и `height` задают, соответственно, ширину и высоту блочного элемента страницы:

```
width: auto|<ширина>
height: auto|<высота>
```

Значение `auto` отдает управление этим размером на откуп Web-обозревателю (обычное поведение). При этом Web-обозреватель установит такие размеры элемента страницы, чтобы в нем полностью поместилось его содержимое, и не больше.

В следующих примерах мы задаем для семантической панели навигации (тега `<nav>`) ширину в 100 пикселов, а для блока `header` высоту в 10% от высоты родителя:

```
nav { width: 100px; }
div#header { height: 10%; }
```

Атрибуты стилей `min-width` и `min-height` устанавливают минимальную ширину и высоту элемента страницы:

```
min-width: <ширина>
min-height: <высота>
```

Пример:

```
table { min-width: 500px;
        min-height: 300px; }
```

Аналогичные атрибуты стилей `max-width` и `max-height` задают максимальную, соответственно, ширину и высоту элемента страницы:

```
max-width: <ширина>
max-height: <высота>
```

Пример:

```
table { max-width: 1000px;
        max-height: 700px; }
```

Напоследок рассмотрим один важный момент. Когда мы указываем размеры элемента страницы с помощью атрибутов стилей `width` и `height`, Web-обозреватель задает эти размеры не самому элементу с учетом заданных для него внутренних отступов и толщины рамки, а его содержимому. Следовательно, реальные размеры элемента будут несколько больше, поскольку к ним добавятся величины внутренних отступов и толщины рамки.

Так, если мы укажем для элемента следующий стиль:

```
div#main { width: 500px;
           padding 10px 20px; }
```

то в реальности ширина блока `main` станет равной 540 пикселам (500 пикселов ширины плюс по 20 пикселов внутренних отступов слева и справа).

Но мы можем упростить себе задачу вычисления и установки размеров, указав в стиле атрибут `box-sizing`. Он определяет, размеры какой части блочного элемента зададут атрибуты стилей, рассмотренные нами ранее:

```
box-sizing: content-box|border-box
```

Доступных значений здесь два:

- `content-box` — рассмотренные ранее атрибуты стилей зададут размер содержимого блочного элемента без учета его внутренних отступов и рамки (поведение по умолчанию);
- `border-box` — рассмотренные ранее атрибуты стилей зададут полные размеры блочного элемента с учетом его внутренних отступов и рамки.

Вот теперь ширина блока `main` будет равна строго 500 пикселов:

```
div#main { width: 500px;
           padding 10px 20px;
           box-sizing: border-box; }
```

Внешние отступы не учитываются при установке размеров элемента в любом случае. Что вполне логично, ведь внешние отступы фактически не принадлежат элементу, для которого установлены.

Параметры переполнения. Элементы с прокруткой

Если мы не зададим для блочного элемента размеры, Web-обозреватель возьмет управление этим параметром на себя. И установленные им размеры будут минимально необходимыми для того, чтобы элемент вместил все свое содержимое.

Но что случится, если для блочного элемента мы зададим слишком маленькие размеры? Тогда может случиться так, что содержимое элемента не поместится в нем, и возникнет *переполнение*. Поведение элемента в этом случае зависит от заданных для него параметров.

Атрибут стиля `overflow` как раз и задает поведение блочного элемента при переполнении:

```
overflow: visible|hidden|scroll|auto
```

Здесь доступны четыре значения:

- `visible` — не помещающееся в элемент содержимое выйдет за его границы (обычное поведение);

- `hidden` — не помещающееся в элемент содержимое будет обрезано, а элемент сохранит свои размеры;
- `scroll` — в элементе появятся полосы прокрутки, с помощью которых станет возможным просмотреть не помещающуюся часть содержимого. Эти полосы прокрутки будут присутствовать в элементе всегда, даже если в них нет нужды;
- `auto` — полосы прокрутки появятся в элементе, только если в них возникнет необходимость.

Атрибут стиля `overflow` имеет смысл только в том случае, если у элемента заданы размеры.

В следующем примере мы задали для блока `main` такое поведение, когда при выходе содержимого за границы блока в нем появятся полосы прокрутки:

```
div#main { height: 500px;
           overflow: auto; }
```

Атрибуты стиля `overflow-x` и `overflow-y` задают поведение блочного элемента при переполнении, соответственно, по горизонтали и вертикали. Доступные значения у них те же, что и у атрибута стиля `overflow`.

Пример:

```
nav { height: 200px;
      overflow-x: hidden; }
```

Пользуясь только что изученными атрибутами стиля, мы можем создать на Web-странице *элементы с прокруткой*. Их преимущество в том, что посетитель, прокручивая содержимое такого элемента, не затрагивает все остальные фрагменты страницы. Весьма удобно.

Здесь нужно описать один трюк, который может очень нам пригодиться. Если какой-либо элемент включает в свой состав плавающие или позиционируемые элементы (о них речь пойдет позже), то может случиться так, что высота этих элементов окажется больше высоты элемента-родителя, и они будут выступать за его пределы. Чтобы растянуть родителя по высоте так, чтобы он полностью вместил все плавающие и позиционируемые элементы-потомки, не указывая его размеры явно, достаточно задать для него атрибут стиля `overflow` или `overflow-y` со значением `hidden`.

Если мы собираемся применять элементы без прокрутки, которые просто обрезают текстовое содержимое, выходящее за их пределы, нам пригодится еще один атрибут стиля. Он носит название `text-overflow` и управляет тем, как Web-обозреватель будет завершать видимую часть текстового содержимого:

```
text-overflow: clip|ellipsis
```

- `clip` — видимая часть текстового содержимого ничем завершаться не будет (значение по умолчанию);
- `ellipsis` — видимая часть будет завершаться многоточием.

Листинг 12.5 указывает для блоков со стилевым классом `announce` завершение видимой части текстового содержимого многоточием.

Листинг 12.5

```
div.announce { width: 100%;
               height: 50px;
               overflow: hidden;
               text-overflow: ellipsis; }
```

Параметры тени у блочного элемента

В *главе 9*, разбираясь с инструментами CSS для указания параметров текста, мы выяснили, что у текста можно создать тень. А есть ли возможность сделать это у самого блочного элемента?

Можно. Требуется лишь указать в соответствующем стиле атрибут `box-shadow`:

```
box-shadow: <горизонтальное смещение> <вертикальное смещение>
[<радиус размытия>] [<расширение>] <цвет> [inset] | none
```

Горизонтальное и *вертикальное* смещения и *радиус размытия* здесь указываются так же и имеют тот же смысл, что и у атрибута стиля `text-shadow` (см. *главу 9*). *Расширение* задает величину, на которую тень увеличится относительно размеров блочного элемента, для которого она задана, — если этот параметр не указан, тень будет иметь те же размеры, что у самого элемента. А языковая конструкция `inset` указывает создать «внутреннюю» тень, визуально находящуюся внутри элемента.

Рассмотрим пару примеров:

- задаем для блока с привязанным стилевым классом `spoiler` черную тень, смещенную на 1 пиксел вправо и вниз, с радиусом размытия в 2 пиксела и расширением в 1 пиксел:

  ```
  div.spoiler { box-shadow: 1px 1px 2px 1px black; }
  ```

- а блоки с привязанным стилевым классом `spoiler2` получат такую же тень, но располагающуюся внутри, а не, как обычно, снаружи:

  ```
  div.spoiler2 { box-shadow: 1px 1px 2px 1px black inset; }
  ```

Параметры выделения

Мы знаем множество способов привлечь внимание посетителя к определенным элементам страниц, использовав теги HTML или атрибуты стиля CSS. Но существует еще один способ сделать это — так называемое *выделение*. Именно о нем сейчас и пойдет речь.

- Выделение CSS 3 представляет собой рамку, которой окружается элемент страницы.

- Выделение, в отличие от знакомой нам рамки CSS, не увеличивает размеры элемента страницы.
- Выделение, в отличие от рамки CSS, можно применять не только к блочным, но и к встроенным элементам страниц.

Для задания параметров выделения предназначены четыре специальных атрибута стилей. Сейчас мы их рассмотрим.

Атрибут стиля `outline-style` задает стиль линий, которыми будет нарисована рамка выделения:

```
outline-style: none|dotted|dashed|solid|double|groove|ridge|inset|outset
```

Значения здесь доступны те же, что и для атрибута стиля `border-style`.

В следующем примере мы задаем для содержимого тега `<strong>` пунктирную рамку выделения:

```
strong { outline-style: dashed; }
```

Атрибут стиля `outline-width` задает толщину рамки выделения:

```
outline-width: thin|medium|thick|<толщина>
```

Здесь доступны те же значения, что и для знакомого нам атрибута стиля `border-width`.

В следующем примере для содержимого тега `<strong>` задана тонкая рамка выделения:

```
strong { outline-style: dashed;
         outline-width: thin; }
```

Атрибут стиля `outline-color` задает цвет рамки выделения:

```
outline-color: <цвет>
```

Пример:

```
strong { outline-width: thin;
         outline-color: grey;
         outline-style: dashed; }
```

Атрибут стиля `outline` позволяет задать сразу все параметры для рамки выделения:

```
outline: <цвет> <стиль> <толщина>
```

Вот пример:

```
strong { outline: thin dashed black; }
```

Параметры таблиц

Среди всех элементов страницы таблицы стоят особняком. Это, вероятно, самый сложный из элементов, включающий в себя множество составных частей: строки, ячейки, секции, заголовок, колонки и группы колонок. Поэтому неудивительно, что

указание параметров таблиц, даже с применением уже знакомых нам атрибутов стилей, имеет некоторые особенности.

Параметры выравнивания

Для выравнивания содержимого ячеек таблицы по горизонтали мы применим атрибут стиля `text-align`, описанный в *главе 9*:

```
td.centered { text-align: center; }
```

Этот же атрибут стиля пригоден для выравнивания текста в заголовке таблицы (теге `<caption>`):

```
caption { text-align: left; }
```

Содержимое ячеек таблиц по вертикали мы выровняем с помощью атрибута стиля `vertical-align`, также описанного в *главе 9*. Однако в случае ячеек таблиц он ведет себя несколько по-другому:

- `top` — выравнивает содержимое ячейки по ее верхнему краю (обычное поведение);
- `middle` — выравнивает содержимое ячейки по ее центру;
- `bottom` — выравнивает содержимое ячейки по ее нижнему краю.

Пример:

```
td, th { vertical-align: middle; }
```

Параметры отступов и рамок

Для задания внутренних отступов в ячейках таблиц мы можем пользоваться атрибутами стилей, описанными в начале этой главы: `padding-left`, `padding-top`, `padding-right`, `padding-bottom` и `padding`.

Для задания величины просветов между ячейками используется атрибут стиля `border-spacing`:

```
border-spacing: <просвет 1> [<просвет 2>]
```

- Если указано одно значение, оно задаст величину просвета со всех сторон ячейки таблицы.
- Если указаны два значения, первое задаст величину просвета слева и справа, а второе — сверху и снизу.

Атрибут стиля `border-spacing` применяется только к таблицам (тегу `<table>`):

```
table { border-spacing: 1px; }
```

Параметры рамок указываются посредством атрибутов стилей, которые также знакомы нам по этой главе.

Показанный в листинге 12.6 код назначает для самой таблицы тонкую сплошную черную рамку и просвет между ячейками в 4 пиксела, а для ячеек этой таблицы —

тонкую пунктирную черную рамку и отступ между границей ячейки и ее содержимым в 4 пиксела.

Листинг 12.6

```
table  { border: medium solid black;
         border-spacing: 4px; }
td, th { border: thin dotted black;
         padding: 4px; }
```

Если мы зададим рамки вокруг ячеек таблицы, Web-обозреватель нарисует рамку вокруг каждой ячейки. Такая таблица будет выглядеть как набор прямоугольников-ячеек, заключенный в большой прямоугольник-таблицу (рис. 12.1).

Столбец 1	Столбец 2	Столбец 3
Ячейка 1.1	Ячейка 1.2	Ячейка 1.3
Ячейка 2.1	Ячейка 2.2	Ячейка 2.3

Рис. 12.1. Обычное поведение Web-обозревателя — рамки рисуются вокруг каждой ячейки таблицы

Однако в печатных изданиях гораздо чаще встречаются таблицы другого вида — в них рамки присутствуют только между ячейками (рис. 12.2).

Столбец 1	Столбец 2	Столбец 3
Ячейка 1.1	Ячейка 1.2	Ячейка 1.3
Ячейка 2.1	Ячейка 2.2	Ячейка 2.3

Рис. 12.2. Таблица, в которой рисуются только рамки, разделяющие ячейки

Атрибут стиля `border-collapse` указывает Web-обозревателю, как следует рисовать рамки ячеек в таблице:

```
border-collapse: separate|collapse
```

- `separate` — каждая ячейка таблицы заключается в отдельную рамку; дополнительно в отдельную рамку заключается сама таблица (см. рис. 12.1; обычное поведение);
- `collapse` — рисуются рамки, разделяющие ячейки таблицы (см. рис. 12.2).

Этот атрибут стиля применяется только к самим таблицам (тегам `<table>`).

Пример:

```
table { border-collapse: collapse; }
```

Параметры размеров

Для задания размеров — ширины и высоты — таблиц и их ячеек подойдут атрибуты стиля `width` и `height`.

- Если требуется задать ширину или высоту всей таблицы, нужный атрибут стиля указывают именно для нее:

```
table { width: 100%;
        height: 300px; }
```

- Если требуется задать ширину столбца, атрибут стиля `width` указывают для первой ячейки, входящей в этот столбец (листинг 12.7).

Листинг 12.7

```
<table>
  <tr>
    <th>Первый столбец</th>
    <th style="width: 40px;">Второй столбец шириной в 40 пикселов</th>
    <th>Третий столбец</th>
  </tr>
  . . .
</table>
```

Можно также указать ширину группы колонок, в состав которой входит нужный столбец. Листинг 12.8 показывает, как это сделать, — второй столбец таблицы будет иметь ширину 40 пикселов.

Листинг 12.8

```
<table>
  <colgroup></colgroup>
  <colgroup style="width: 50px;"></colgroup>
  <colgroup></colgroup>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
  </tr>
  . . .
</table>
```

Можно указывать ширину и колонки, и группы колонок. Так, в листинге 12.9 показан код, задающий совокупную ширину первой группы колонок в 100 пикселов, при этом первый столбец, входящий в эту группу, получит ширину 30 пикселов.

Листинг 12.9

```
<table>
  <colgroup style="width: 100px;">
    <col style="width: 30px;">
    <col span="2">
  </colgroup>
  <tr>
    <th>Столбец 1</th>
    <th>Столбец 2</th>
    <th>Столбец 3</th>
  </tr>
  . . .
</table>
```

- Если требуется задать высоту строки, атрибут стиля `height` указывают для первой ячейки этой строки (листинг 12.10).

Листинг 12.10

```
<table>
  . . .
  <tr>
    <td style="height: 30px;">Строка высотой в 30 пикселов</td>
    . . .
  </tr>
  . . .
</table>
```

По умолчанию все размеры, которые мы зададим для таблицы и ее ячеек, — не более чем рекомендация для Web-обозревателя. Если содержимое таблицы не будет в ней помещаться, Web-обозреватель увеличит ширину или высоту таблицы. Зачастую это может быть неприемлемо, поэтому стандарт CSS предусматривает средства, позволяющие изменить такое поведение.

Атрибут стиля `table-layout` дает возможность указать, как Web-обозреватель будет трактовать заданные нами для таблицы и ее ячеек размеры:

```
table-layout: auto|fixed
```

- `auto` — Web-обозреватель может изменить размеры таблицы и ее ячеек, если содержимое в них не помещается (поведение по умолчанию).
- `fixed` — размеры таблицы и ее ячеек ни в коем случае изменяться не будут. Если содержимое в них не помещается, возникнет переполнение, параметры которого мы можем задавать с помощью атрибутов стилей `overflow`, `overflow-x` и `overflow-y`.

Этот атрибут стиля применяется к самой таблице (тегу `<table>`).

Пример:

```
table { table-layout: fixed;
        overflow: auto; }
```

Прочие параметры

И еще несколько полезных атрибутов стилей, которые помогут нам задать дополнительные параметры таблиц.

Атрибут стиля `caption-side` указывает местоположение заголовка таблицы относительно самой таблицы:

```
caption-side: top|bottom
```

- `top` — заголовок располагается над таблицей (обычное поведение).
- `bottom` — заголовок располагается под таблицей.

Этот атрибут стиля применяется к самой таблице (тегу `<table>`).

Пример:

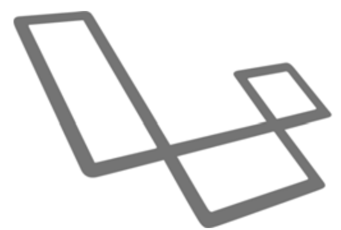
```
table { caption-side: bottom; }
```

Атрибут стиля `empty-cells` указывает, как Web-обозреватель должен выводить на экран пустые (не имеющие содержимого) ячейки:

```
empty-cells: show|hide
```

- `show` — пустые ячейки будут выводиться на экран. Если для них был задан другой фон, на экран будет выведен фон, а если заданы рамки, будут выведены рамки. Это поведение по умолчанию.
- `hide` — пустые ячейки вообще не будут выводиться на экран.

Атрибут стиля `empty-cells` также применяется к самой таблице (тегу `<table>`).



ГЛАВА 13

Инструменты для создания разметки

Все атрибуты стилей, с которыми мы познакомились в предыдущих главах этого раздела, предназначены для оформления элементов страниц. Они указывают шрифт, которым будет выводиться текст, задают для него фон, устанавливают размеры, отступы и рамку, даже могут скрыть элемент — но не более.

Но для того чтобы создать разметку страницы, нужно нечто большее. Необходимы такие атрибуты стилей, которые позволят нам разместить элементы в нужных местах страницы, — туда, куда их соблаговолил поставить Web-дизайнер, по чьему макету мы верстаем страницу.

Что такое разметка Web-страницы? Мы узнаем это потом. Сначала следует познакомиться с атрибутами стилей, применяемыми для ее создания, и заодно с новыми разновидностями элементов страницы.

Плавающие элементы

Как мы выяснили еще в *главе 2*, все блочные элементы выводятся друг за другом по вертикали и занимают всю доступную ширину родительского элемента (другого блочного элемента или самой страницы). Это их поведение по умолчанию.

Но мы можем сделать так, чтобы какой-либо блочный элемент:

- сдвигался к левому или правому краю родителя, а все остальное содержимое последнего обтекало его;
- получал такую ширину, чтобы только вместить свое содержимое.

Иными словами — создать *плавающий элемент*.

Для создания плавающих элементов служит атрибут стиля `float`. Он указывает, к какому краю родительского элемента должен сдвигаться элемент страницы:

```
float: left|right|none
```

- `left` — элемент страницы сдвигается к левому краю родительского элемента, а остальное содержимое обтекает его справа.

- `right` — элемент страницы сдвигается к правому краю родительского элемента, а остальное содержимое обтекает его слева.
- `none` — элемент страницы ведет себя как обычно (поведение по умолчанию).

В следующем примере мы превращаем семантическую «врезку» в плавающий элемент страницы, который сдвинется к правому краю родителя (рис. 13.1):

```
aside { float: right; }
```

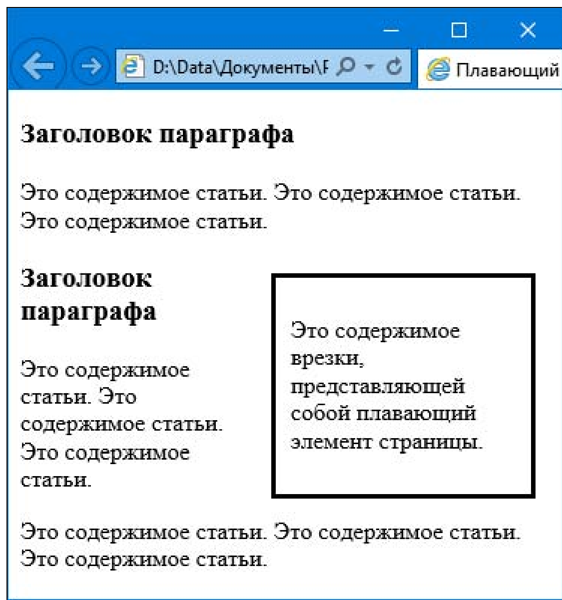


Рис. 13.1. Плавающий элемент страницы

А что, если мы зададим одинаковое значение атрибута стиля `float` для нескольких следующих друг за другом блочных элементов? Они выстроятся по горизонтали друг за другом в том порядке, в котором они определены в HTML-коде, и будут сдвинуты к указанному краю родителя. Это можно использовать, чтобы выстроить блочные элементы по горизонтали, однако сейчас для этой цели удобнее применять гибкую верстку (о которой мы обязательно поговорим).

Но что, если нам понадобится сделать так, чтобы содержимое какого-либо блочного элемента, — например, заголовка, не обтекало плавающий, а было выведено под ним? Указать для него стиль с атрибутом `clear`:

```
clear: left|right|both|none
```

Доступных значений здесь четыре:

- `left` — элемент страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `left`;
- `right` — элемент страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `right`;

- `both` — элемент страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `left` или `right`;
- `none` — обычное поведение (значение по умолчанию).

В следующем примере мы задали для заголовков третьего уровня расположение ниже любых предшествующих ему плавающих элементов (результат можно увидеть на рис. 13.2):

```
h3 { clear: both; }
```

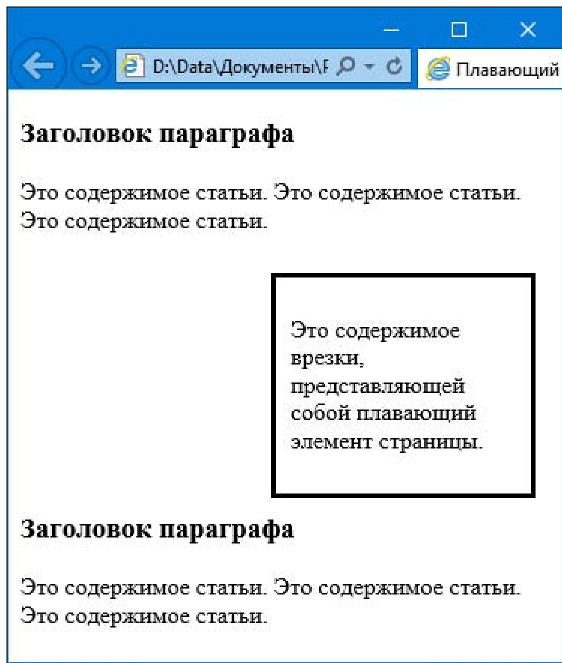


Рис. 13.2. Заголовок выводится ниже плавающего элемента страницы, а не обтекает его

Позиционируемые элементы

Для плавающих элементов мы можем лишь указать край родителя, к которому должен быть сдвинут элемент. Для позиционируемых элементов мы можем задать произвольное местоположение.

Понятие позиционируемого элемента

Web-обозреватель обрабатывает элементы страницы в том порядке, в котором в ее HTML-коде расположены создающие их теги. И выводит их на экран он точно в таком же порядке — в направлении слева направо и сверху вниз.

Управлять местоположением элементов страницы в этом случае мы не можем. Поэтому такие элементы называются *непозиционируемыми*.

Но довольно часто возникает необходимость задать для какого-либо элемента произвольное местоположение. Причем сделать так, чтобы этот элемент выводился поверх остального содержания страницы, перекрывая его. Таким элементом может быть, например, всплывающее окошко, в котором выводится полная версия изображения, — так называемый лайтбокс.

Для таких случаев CSS предоставляет нам средства создания *позиционируемых* элементов. Рассмотрим их особенности и отличия от непозиционируемых элементов.

- Местоположение позиционируемого элемента задается произвольно в виде горизонтальной и вертикальной координат его верхнего левого угла.
- Под позиционируемый элемент на странице место не выделяется.
- Позиционируемые элементы находятся «выше» обычного содержания страницы, как бы «плавают» над ним и перекрывают его.
- Позиционируемые элементы могут перекрывать друг друга.
- Позиционируемые элементы могут иметь любое содержимое, в том числе и другие позиционируемые элементы.

Ранее стандарт CSS позволял сделать позиционируемыми только блочные контейнеры. Теперь же мы можем превратить в позиционируемый также элемент семантической разметки любого типа (`<header>`, `<nav>`, `<section>`, `<footer>` и др.).

Создание позиционируемых элементов

Позиционируемые элементы страниц создаются с помощью особых атрибутов стилей CSS, которые мы сейчас рассмотрим.

Самый важный атрибут стиля — `position`. Он задает способ позиционирования элемента страницы:

```
position: static|absolute|relative|fixed
```

Этот атрибут стиля может принимать четыре значения:

- `static` — превращает элемент в непозиционируемый (поведение по умолчанию);
- `absolute` — превращает элемент в *свободно позиционируемый*. Координаты такого элемента задаются относительно левого верхнего угла ближайшего позиционируемого родителя или, если такового нет, самой страницы. Если содержимое родителя прокручивается, свободно позиционируемый элемент будет перемещаться вместе с ним;
- `relative` — превращает элемент в *относительно позиционируемый*. Координаты такого элемента отсчитываются относительно точки, в которой он находился бы, если был непозиционируемым. Если содержимое родителя прокручивается, относительно позиционируемый элемент будет перемещаться вместе с ним;
- `fixed` — превращает элемент в *фиксированный*. Он ведет себя как свободно позиционируемый элемент, за тем исключением, что его координаты задаются

относительно левого верхнего угла клиентской области окна Web-обозревателя. Следовательно, при прокрутке содержания страницы фиксированный элемент всегда будет оставаться на месте.

В следующем примере мы превратили блок `box` в свободно позиционируемый:

```
div#box { position: absolute; }
```

Для указания местоположения позиционируемого элемента применяются следующие атрибуты стилей:

- `left` — горизонтальная координата левого верхнего угла элемента;
- `top` — вертикальная координата левого верхнего угла элемента;
- `right` — горизонтальная координата правого нижнего угла элемента;
- `bottom` — вертикальная координата правого нижнего угла элемента.

Они записываются в следующем формате:

```
left|top|right|bottom: <значение>|auto
```

Значение `auto` возвращает управление соответствующей координатой Web-обозревателю. Как правило, он устанавливает соответствующую координату равной нулю.

В примере из листинга 13.1 мы превратили блок `box1` в свободно позиционируемый элемент и задали для него местоположение посредством указанных ранее атрибутов стилей.

Листинг 13.1

```
div#box1 { position: absolute;
           left: 200px;
           top: 100px;
           right: 100px;
           bottom: 200px; }
```

Другой вариант — указать местоположение одного из углов элемента и его размеры. Листинг 13.2 демонстрирует стиль подобного рода.

Листинг 13.2

```
div#box2 { position: absolute;
           left: 200px;
           top: 100px;
           width: 500px;
           height: 100px; }
```

Ранее говорилось, что позиционируемые элементы могут перекрывать друг друга. При этом элемент, определенный в HTML-коде позже, перекрывает элемент, определенный раньше. Однако мы можем сами задать порядок их перекрытия друг дру-

гом, указав так называемый *z-индекс*. Он представляет собой целое число, указывающее номер в порядке перекрытия, — при этом элементы с бóльшим z-индексом перекрывают элементы с меньшим z-индексом. Z-индекс задается атрибутом стиля с «говорящим» именем `z-index`:

```
z-index: <номер>|auto
```

Повторим: z-индекс указывается в виде целого числа. Значение `auto` возвращает управление порядком перекрытия Web-обозревателю. В листинге 13.3 приведен пример использования z-индекса — блок `over` перекроет блок `under`, т. к. для него задан больший z-индекс.

Листинг 13.3

```
div#over  { position: absolute;
            left: 200px;
            top: 100px;
            width: 300px;
            height: 200px;
            z-index: 2; }
div#under { position: absolute;
            left: 100px;
            top: 0px;
            width: 600px;
            height: 500px;
            z-index: 0; }
```

Еще один атрибут стиля, который иногда может быть полезен, — `clip`. Он определяет координаты прямоугольной области, задающей видимую область позиционируемого элемента. Фрагмент содержимого элемента, попадающий в эту область (ее, кстати, называют *маской*), будет видим на Web-странице, остальная часть содержимого окажется скрытой.

Вот формат записи атрибута стиля `clip`:

```
clip: rect(<верхняя граница>, <правая граница>, <нижняя граница>,
<левая граница>) | auto
```

Здесь:

- *верхняя граница* — расстояние от верхней границы позиционируемого элемента до верхней границы маски по вертикали;
- *правая граница* — расстояние от левой границы позиционируемого элемента до правой границы маски по горизонтали;
- *нижняя граница* — расстояние от верхней границы позиционируемого элемента до нижней границы маски по вертикали;
- *левая граница* — расстояние от левой границы позиционируемого элемента до левой границы маски по горизонтали.

Значение `auto` атрибута стиля `clip` убирает маску и тем самым делает все содержимое позиционируемого элемента видимым. Это поведение по умолчанию. В листинге 13.4 представлен пример использования атрибута стиля `clip`.

Листинг 13.4

```
div#masked { position: absolute;
             left: 200px;
             top: 100px;
             width: 300px;
             height: 200px;
             clip: rect(100px, 200px, 200px, 0px); }
```

Гибкая верстка

Очень часто на странице требуется расположить своего рода перечень каких-либо элементов-позиций. Причем эти позиции требуется выстраивать в строки по горизонтали, но, если они не помещаются по ширине в одну строку, следует выполнить их перенос.

Ранее для этого приходилось превращать позиции такого перечня в плавающие элементы и долго подбирать для них величины размеров и внешних отступов, чтобы получить нужный результат. Но теперь, в эпоху победного шествия CSS 3, подобного рода перечни реализуются значительно проще — с помощью гибкой верстки.

Гибкая верстка (flex layout) служит для создания элементов-перечней, которые:

- вмещают в себя набор элементов-позиций;
- устанавливают для элементов-позиций одинаковые размеры;
- при необходимости адаптируют ширину или высоту позиций таким образом, чтобы они помещались в элементе-перечне;
- выстраивают позиции либо по строкам, либо по столбцам;
- если активирована такая возможность, переносят позиции на следующую строку или столбец при нехватке места в элементе-перечне;
- позволяют задавать выравнивание как сразу всех позиций в перечне, так и каждой конкретной позиции.

Реализация гибкой верстки

Реализовать гибкую верстку очень просто. Достаточно указать в стиле, который действует на сам элемент-перечень, атрибут стиля `display` со значением `flex`.

В листинге 13.5 приведен пример реализации простейшей гибкой верстки. Для наглядности в элементы-позиции были помещены порядковые номера, а для элемента-перечня и позиций заданы рамки. На рис. 13.3 можно увидеть результат, показанный Web-обозревателем.

Листинг 13.5

```
.flex-cont      { display: flex;
                  border: Black 2px dotted; }
.flex-cont div { font-size: 40pt;
                  text-align: center;
                  width: 100px;
                  margin: 10px;
                  border: Black 1px solid; }
. . .
<div class="flex-cont">
  <div>1</div>
  <div>2</div>
  <div>3</div>
  <div>4</div>
</div>
```

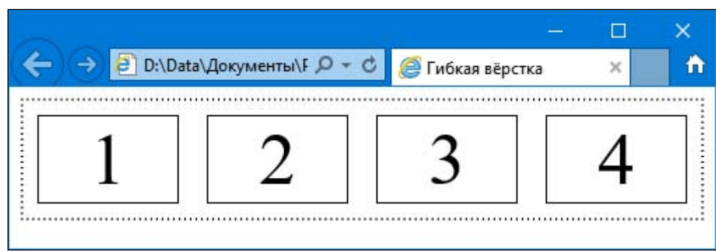


Рис. 13.3. Простейший пример гибкой верстки

Атрибут стиля `flex-direction` управляет направлением выстраивания позиций в перечне:

```
flex-direction: row|row-reverse|column|column-reverse
```

- `row` — позиции выстраиваются в строку, слева направо (поведение по умолчанию);
- `row-reverse` — позиции выстраиваются в строку, справа налево;
- `column` — позиции выстраиваются в столбец, сверху вниз;
- `column-reverse` — позиции выстраиваются в столбец, снизу вверх.

Этот атрибут стиля также указывается для элемента-перечня, например:

```
.flex-cont { flex-direction: column; }
```

По умолчанию Web-обозреватель не выполняет перенос позиций по строкам или столбцам, если позиции не помещаются в элемент-перечень. Вместо этого он уменьшает размеры элементов-позиций таким образом, чтобы они поместились в перечень.

Однако мы можем указать ему выполнять перенос, задав подходящее значение для атрибута стиля `flex-wrap`:

```
flex-wrap: nowrap|wrap|wrap-reverse
```

- `nowrap` — перенос не выполняется (поведение по умолчанию);
- `wrap` — перенос выполняется, при этом строки выстраиваются сверху вниз, а столбцы — слева направо;
- `wrap-reverse` — перенос выполняется, при этом строки выстраиваются снизу вверх, а столбцы — справа налево.

И этот атрибут стиля указывается для элемента-перечня, например (рис. 13.4):

```
.flex-cont { flex-wrap: wrap; }
```

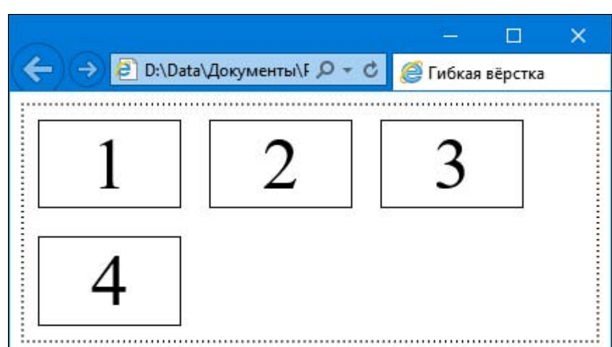


Рис. 13.4. Гибкая верстка с переносом позиций

Выравнивание позиций

Есть возможность указать выравнивание как самих позиций по разным направлениям, так и выравнивание их совокупности (назовем ее *блоком позиций*) в элементе-перечне.

Атрибут стиля `justify-content` задает выравнивание позиций по направлению их выстраивания в перечне (это направление указывается атрибутом стиля `flex-direction`, изученным нами ранее):

```
justify-content: flex-start|flex-end|center|space-between|space-around
```

- `flex-start` — позиции выравниваются по началу направления их выстраивания (например, если было указано направление выстраивания `row`, позиции будут выравниваться по левому краю перечня, а если было указано `column-reverse` — то по нижнему краю). Это поведение по умолчанию;
- `flex-end` — позиции выравниваются по концу направления их выстраивания;
- `center` — позиции выравниваются по середине элемента-перечня;
- `space-between` — между позициями вставляются просветы, чтобы они равномерно распределились по всему пространству элемента-перечня;
- `space-around` — между позициями, а также между ними и границами элемента-перечня вставляются просветы, чтобы позиции равномерно распределились по всему пространству элемента-перечня.

Этот атрибут стиля указывается для элемента-перечня, например (рис. 13.5):

```
.flex-cont { justify-content: center; }
```

Как видим, выравнивание по направлению выстраивания будет иметь место только в том случае, если в элементе-перечне есть свободное пространство.

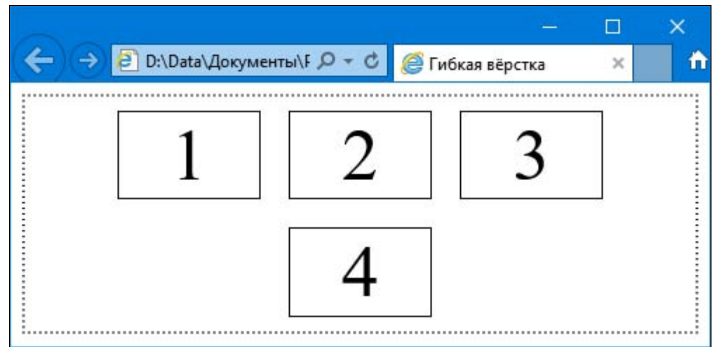


Рис. 13.5. Гибкая верстка с центрированием позиций по направлению их выстраивания

Атрибут стиля `align-items` устанавливает выравнивание позиций в направлении, перпендикулярном направлению их выстраивания:

```
align-items: flex-start|baseline|flex-end|center|stretch
```

- `flex-start` и `baseline` — позиции выравниваются по началу этого направления;
- `flex-end` — позиции выравниваются по концу этого направления;
- `center` — позиции выравниваются посередине в этом направлении;
- `stretch` — если размер позиций в направлении, перпендикулярном направлению их выстраивания, не задан, они растягиваются в этом направлении на все пространство элемента-перечня. Если же размер позиций задан, они будут выровнены по началу этого направления. Это поведение по умолчанию.

Этот атрибут стиля указывается для элемента-перечня.

Выравнивание позиций в направлении, перпендикулярном направлению их выстраивания, имеет смысл задавать лишь тогда, когда в перечне есть свободное пространство. В листинге 13.6 показан стиль, задающий для перечня достаточно большую высоту и выравнивание позиций по концу направления, а на рис. 13.6 представлен результат.

Листинг 13.6

```
.flex-cont { display: flex;
             height: 200px;
             border: Black 2px dotted;
             align-items: flex-end; }
```

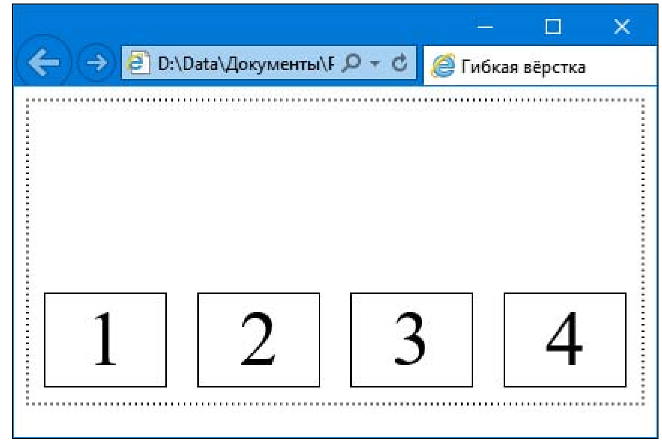


Рис. 13.6. Гибкая верстка с выравниванием позиций по концу направления, перпендикулярном направлению их выстраивания

И, наконец, атрибут стиля `align-content` позволяет задать выравнивание самого блока позиций:

```
align-content: flex-start|flex-end|center|stretch|space-between|
space-around
```

- `flex-start` — блок позиций выравнивается по началу направления, по которому выстраиваются отдельные строки или столбцы;
- `flex-end` — блок позиций выравнивается по концу направления, по которому выстраиваются отдельные строки или столбцы;
- `center` — блок позиций выравнивается посередине в направлении, по которому выстраиваются отдельные строки или столбцы;
- `stretch` — блок позиций растягивается на все пространство элемента-перечня по вертикали, если используется выстраивание по строкам, или по горизонтали, если задействовано выстраивание по столбцам (поведение по умолчанию);
- `space-between` — между строками или столбцами вставляются просветы, чтобы они равномерно распределились по всему пространству элемента-перечня;
- `space-around` — между строками или столбцами, а также между ними и границами элемента-перечня вставляются просветы, чтобы позиции равномерно распределились по всему пространству перечня.

И этот атрибут стиля указывается для элемента перечня. Его имеет смысл задавать только в том случае, если указан также и перенос позиций.

Пример:

```
.flex-cont { flex-wrap: wrap;
             align-content: center; }
```

Выравнивание отдельных позиций

Еще имеется возможность задать выравнивание для отдельных позиций перечня как в направлении их выстраивания, так и в перпендикулярном ему.

Для выравнивания позиций в направлении выстраивания применяются те же атрибуты стилей, что используются для указания внешних отступов (см. *главу 12*).

- Если требуется сдвинуть первую позицию в строке к левому краю перечня, а следующие за ней — к правому, следует указать для первой позиции атрибут стиля `margin-right` со значением `auto`.
- Если требуется сдвинуть последнюю позицию в строке к правому краю перечня, а предыдущие — к левому, следует указать для последней позиции атрибут стиля `margin-left` со значением `auto`.
- Если требуется сдвинуть первую позицию в столбце к верхнему краю перечня, а следующие за ней — к нижнему, следует указать для первой позиции атрибут стиля `margin-bottom` со значением `auto`.
- Если требуется сдвинуть последнюю позицию в столбце к нижнему краю перечня, а предыдущие — к верхнему, следует указать для последней позиции атрибут стиля `margin-top` со значением `auto`.
- Если требуется расположить какую-либо позицию в середине перечня по горизонтали, следует указать для нее атрибуты стилей `margin-left` и `margin-right` со значениями `auto`.
- Если требуется расположить какую-либо позицию в середине перечня по вертикали, следует указать для нее атрибуты стиля `margin-top` и `margin-bottom` со значениями `auto`.
- Если требуется расположить один-единственный элемент строго в середине родителя и по горизонтали, и по вертикали, следует указать для родителя гибкую верстку, а для самого элемента — атрибут стиля `margin` со значением `auto`.

Соответствующие стили привязываются непосредственно к позициям, которые следует выровнять.

В следующем примере мы сдвигаем первую позицию в перечне влево, а следующие за ней в той же строке смещаем вправо:

```
.flex-cont div:first-child { margin-right: auto; }
```

На рис. 13.7 показан элемент-перечень, в котором для некоторых позиций было указано различное выравнивание в направлении их выстраивания. Необходимые атрибуты стилей и их значения написаны непосредственно на выравниваемых позициях.

Если же нужно выровнять позиции в направлении, противоположном направлению их выстраивания, нам на помощь придет атрибут стиля `align-self`:

```
align-self: auto|flex-start|baseline|flex-end|center|stretch
```

- `auto` — выравнивание устанавливается атрибутом стиля `align-items`, описанным ранее и заданным у самого элемента-перечня (поведение по умолчанию);
- `flex-start` и `baseline` — позиция выравнивается по началу этого направления;
- `flex-end` — позиция выравнивается по концу этого направления;
- `center` — позиция выравнивается посередине в этом направлении;
- `stretch` — если размер позиции в направлении, перпендикулярном направлению выстраивания, не задан, она растянется в этом направлении на все пространство элемента-перечня. Если же размер позиции задан, она будут выровнена по началу этого направления.

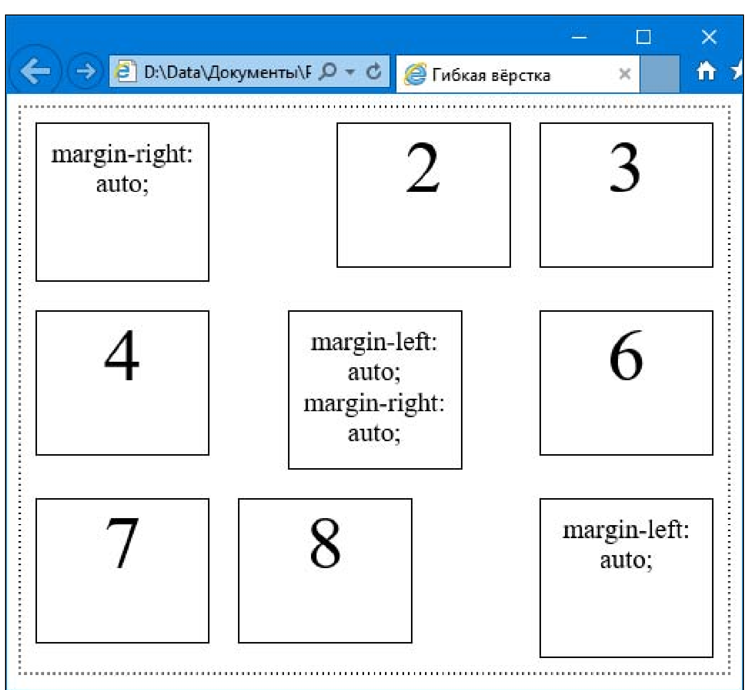


Рис. 13.7. Пример выравнивания элементов-позиций в направлении их выстраивания

В следующем примере мы выравниваем четные позиции по началу направления, а нечетные — по его концу:

```
.flex-cont div:nth-child(even) { align-self: flex-start; }
.flex-cont div:nth-child(odd)  { align-self: flex-end; }
```

На рис. 13.8 представлен элемент-перечень, в котором для некоторых позиций было указано различное выравнивание в направлении, перпендикулярном направлению их выстраивания. Необходимые атрибуты стилей и их значения написаны непосредственно на выравниваемых позициях.

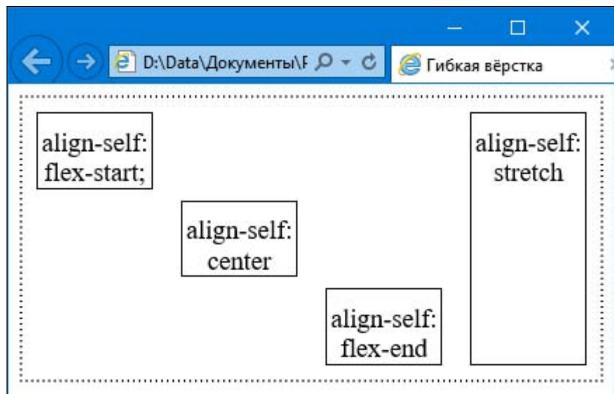


Рис. 13.8. Пример выравнивания элементов-позиций в направлении, перпендикулярном направлению их выстраивания

Управление размерами и порядком следования позиций

По умолчанию, если в элементе-перечне есть свободное пространство в направлении выстраивания позиций, последние не будут увеличиваться в размерах, чтобы занять его полностью. Однако, если в этом направлении, напротив, не хватает места и при этом не задействован перенос, позиции будут пропорционально уменьшены, чтобы поместиться в перечень.

Это поведение мы можем изменить, использовав четыре атрибута стилей, которые сейчас рассмотрим. Все эти атрибуты стилей указываются непосредственно для позиций.

- Атрибут стиля `flex-basis` задает изначальный размер позиций в направлении их выстраивания. Так, если позиции выстраиваются в строку, он задает их ширину, а если в столбец — высоту:

  `flex-basis: auto|<`*`размер`*`>`

  Значение `auto` либо установит для позиций размер, заданный атрибутами стилей `width` и `height`, либо, если таковые отсутствуют, передаст управление размерами элементов Web-обозревателю.

- Атрибут стиля `flex-grow` указывает, насколько сильнее текущая позиция будет увеличиваться относительно других позиций, чтобы заполнить оставшееся в перечне свободное пространство:

  `flex-grow: <степень увеличения>`

  `Степень увеличения` указывается в виде числа без единицы измерения. Если задать значение `0`, позиция не будет увеличиваться. Значение по умолчанию — `0`.

- Атрибут стиля `flex-shrink` указывает, насколько сильнее текущая позиция будет уменьшаться относительно других позиций, чтобы поместиться в перечень при нехватке в нем места:

  `flex-shrink: <степень уменьшения>`

> `Степень уменьшения` также задается в виде числа без единицы измерения. Если задать значение 0, позиция не будет уменьшаться. Значение по умолчанию — 1 (позиция уменьшается в такой же степени, что и остальные).

В листинге 13.7 приведены стили, задающие для всех позиций изначальный размер по направлению выстраивания 100 пикселов и указывающие последней позиции увеличиваться, чтобы занять свободное пространство, а при его нехватке — уменьшаться вдвое относительно остальных. Результаты применения этих стилей показаны на рис. 13.9 и 13.10.

Листинг 13.7

```
.flex-cont div             { flex-basis: 100px; }
.flex-cont div:last-child { flex-grow: 1;
                             flex-shrink: 2; }
```

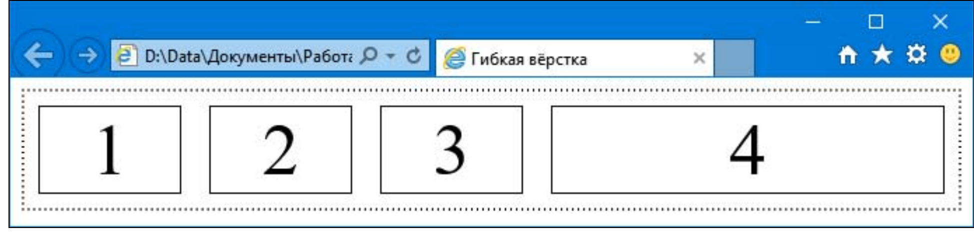


Рис. 13.9. Пример управления размерами отдельной позиции перечня: последняя позиция увеличилась, чтобы занять свободное пространство в перечне

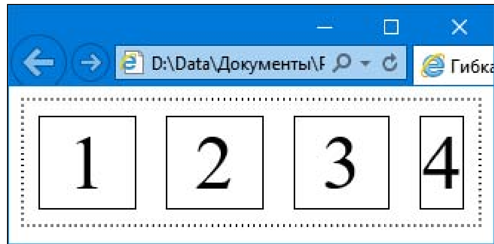


Рис. 13.10. Пример управления размерами отдельной позиции перечня: последняя позиция при нехватке места в перечне уменьшилась вдвое относительно остальных

Атрибут стиля `flex` позволяет указать сразу все три перечисленных ранее параметра:

```
flex: auto | none | <величина увеличения> <величина уменьшения>
<изначальный размер>
```

Если задано значение `auto`, изначальный размер позиции задается либо атрибутами стилей `width` и `height`, либо Web-обозревателем, — позиции при этом увеличиваются и уменьшаются в одинаковой степени (то же, что `1 1 auto`). Если задано значение `none`, изначальный размер элемента задается либо атрибутами стилей `width` и

`height`, либо Web-обозревателем, — позиции при этом не увеличиваются и не уменьшаются (то же, что `0 0 auto`).

В следующем примере мы указываем для всех позиций изначальный размер 100 пикселов и одинаковую степень увеличения и уменьшения:

```
.flex-cont div { flex: 1 1 100px; }
```

По умолчанию позиции выводятся в перечне в том порядке, в котором они присутствуют в HTML-коде страницы. Но мы можем изменить этот порядок.

Атрибут стиля `order` указывает для позиции номер в порядке следования:

```
order: <номер в порядке следования>
```

Номер в порядке следования указывается в виде числа, допускаются положительные, отрицательные числа и 0. Позиции выстраиваются в порядке увеличения заданных для них номеров. Значение по умолчанию — 0.

В следующем примере мы выводим первую позицию в самом конце перечня, а последнюю — в самом его начале:

```
.flex-cont div:first-child { order: 10; }
.flex-cont div:last-child  { order: -1; }
```

Многоколоночная верстка

CSS 3 принес в Web-верстку инструменты для вывода текста в несколько колонок. Это довольно спорное нововведение: Web-страница — не страница книги или журнала, она имеет практически неограниченную высоту, и просматривать ее удобнее, прокручивая содержимое вверх. Если же вывести ее содержание в несколько колонок, посетителю, чтобы ознакомиться с ним, придется после прочтения очередной колонки прокручивать страницу вниз, возвращаясь к ее началу, что крайне неудобно.

На взгляд автора, многоколоночная верстка уместна в двух случаях. Во-первых, при верстке небольших фрагментов текста, которые гарантированно поместятся в клиентской области окна Web-обозревателя. Во-вторых, если многоколоночный текст помещен в элемент с горизонтальной прокруткой (как создаются элементы с прокруткой, было описано в *главе 12*).

Базовые средства многоколоночной верстки

Если нужно просто вывести текст в несколько колонок, достаточно применить один из описанных далее атрибутов стилей. (Применять их совместно смысла нет, поскольку они задают взаимоисключающие параметры.)

Оба этих атрибута стилей указываются для элемента, содержимое которого требуется вывести в несколько колонок.

- Атрибут стиля `column-count` задает количество колонок:

  ```
  column-count: <количество колонок>|auto
  ```

`Количество колонок` указывается числом без единицы измерения. Если задано значение `auto`, содержимое будет выведено в одну колонку (это поведение по умолчанию).

Что касается ширины получившихся колонок, то она будет установлена Web-обозревателем таким образом, чтобы заполнить ими всю ширину элемента.

В следующем примере мы выводим содержимое семантической статьи (тега `<article>`) с привязанным стилевым классом `multicolumn1` в две колонки:

```
article.multicolumn1 { column-count: 2; }
```

- Атрибут стиля `column-width` указывает ширину отдельной колонки:

  ```
  column-width: <ширина колонки>|auto
  ```

  Если задано значение `auto`, содержимое будет выведено в одну колонку (поведение по умолчанию).

  Количество колонок в этом случае будет подобрано самим Web-обозревателем таким образом, чтобы, опять же, занять ими всю ширину элемента.

  В следующем примере мы выводим содержимое семантической статьи с привязанным стилевым классом `multicolumn2` в колонки шириной 200 пикселов:

  ```
  article.multicolumn2 { column-width: 200px; }
  ```

Задание дополнительных параметров колонок

Мы можем задать для колонок дополнительные параметры: ширину просвета между ними и внешний вид разделяющей их линии. Еще мы можем сделать так, чтобы какой-то фрагмент содержимого при выводе растягивался на все колонки.

Атрибут стиля `column-gap` устанавливает ширину просвета между колонками:

```
column-gap: <ширина просвета>|normal
```

Значение `normal` передает управление просветом между колонками Web-обозревателю.

В следующем примере мы задаем между колонками двухсантиметровый просвет:

```
article { column-gap: 2cm; }
```

Параметры разделяющей колонки линии указываются с применением следующих атрибутов стилей:

- `column-rule-style` — стиль линий (сплошная, пунктирная, штриховая и пр.);
- `column-rule-width` — толщина линий;
- `column-rule-color` — цвет линий.

Эти атрибуты стилей принимают те же значения, что аналогичные атрибуты, отвечающие за параметры рамки и описанные в *главе 12*.

В листинге 13.8 приведен стиль, задающий для семантической статьи тонкие черные пунктирные линии разделителей.

Листинг 13.8

```
article { column-rule-style: dotted;
          column-rule-width: thin;
          column-rule-color: black; }
```

Атрибут стиля `column-rule` позволяет указать сразу все параметры разделителей:

```
article { column-rule: thin dotted black; }
```

Иногда возникает необходимость сделать так, чтобы какой-то элемент, например заголовок, растягивался сразу на все колонки, как показано на рис. 13.11.

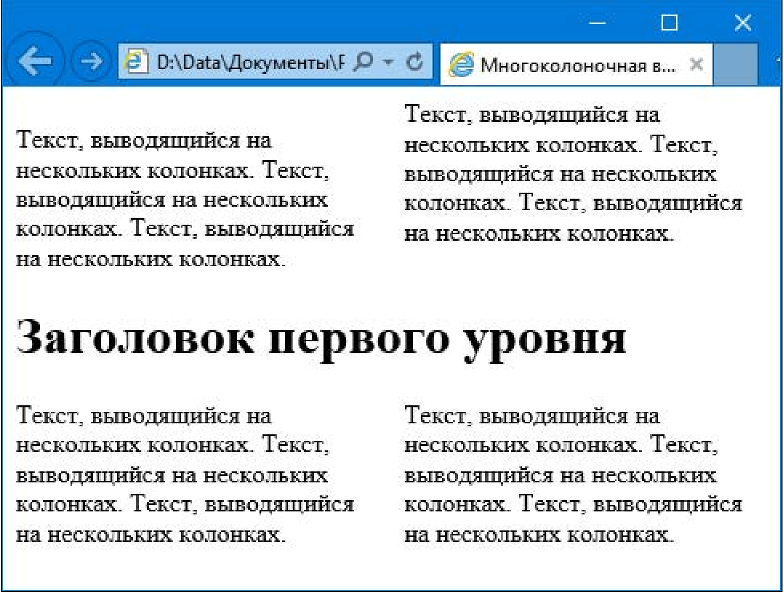


Рис. 13.11. Заголовок, растянутый на все колонки

Атрибут стиля `column-span` указывает, будет ли элемент страницы растягиваться на все колонки:

```
column-span: 1|all
```

Значение `1` заставляет элемент выводиться в колонке, как обычно (это поведение по умолчанию). Значение `all` предписывает ему растянуться на все колонки.

Этот атрибут стиля указывается непосредственно для элемента, который требуется растянуть на несколько колонок.

Пример:

```
h1 { column-span: all; }
```

Разметка Web-страницы и ее создание

При верстке сложных страниц их содержание практически всегда разбивается на отдельные фрагменты: «шапку», панель навигации, основное содержание (уникальное для каждой страницы) и «поддон». Содержимое этих фрагментов помещается в отдельные элементы, которые затем располагаются в нужных местах страницы, в результате чего страница становится существенно удобнее для восприятия.

Совокупность этих элементов вместе с их представлением (которое указывает для них местоположение, размеры и прочее оформление) носит название *разметки*.

Давайте рассмотрим классическую разметку, представленную на рис. 13.12. Там мы видим «шапку» страницы с заголовком сайта и, возможно, дополнительной панелью навигации, расположенную слева основную панель навигации по сайту, «поддон» со сведениями об авторских правах и основное содержание, которое занимает все оставшееся место.

"Шапка"

Основное содержание

Панель навигации

"Поддон"

Рис. 13.12. Классическая разметка страницы

Для создания отдельных элементов, составляющих разметку, удобнее применять теги семантической разметки, рассмотренные в *главе 2*:

- `<header>` — для «шапки»;
- `<nav>` — для основной панели навигации;
- `<section>` или `<article>` — для основного содержания;
- `<footer>` — для «поддона».

Также возможно применение обычных блочных контейнеров (`<div>`) — в основном, для создания вспомогательных элементов.

Что касается представления элементов разметки, то оно может быть основано на разных принципах и, соответственно, быть самым различным. В этом разделе главы мы познакомимся с наиболее перспективными в настоящее время вариантами разметки.

Табличная разметка на основе блоков

Основная проблема при создании разметки, наподобие той, что представлена на рис. 13.12, заключается в том, что следует вывести панель навигации и основное содержание не друг под другом, а по горизонтали. Для этого можно применить три подхода: таблицы, плавающие элементы и элементы с измененным отображением.

Лет десять назад такого рода разметку практически всегда создавали на основе таблиц HTML (см. *главу 4*). Однако разработка ее была весьма трудоемкой, а получающийся в результате HTML-код — слишком громоздким и трудночитаемым. Тем не менее, подобная разметка до сих пор носит название *табличной*.

Применение плавающих элементов позволяет радикально уменьшить и упростить HTML-код. Однако понадобится много времени для создания представления, ведь придется долго подбирать размеры элементов и величины отступов, чтобы панель навигации и основное содержание выводились по горизонтали, а не друг под другом. К тому же, нет простого способа указать для таких элементов одинаковую высоту.

Поэтому много удобнее применять для табличной разметки блочные элементы, для которых задано специальное отображение. Это отображение указывается с применением атрибута стиля `display`, описанного в *главе 10*.

Создается такая разметка следующим образом:

- создается элемент, для которого указывается отображение в виде таблицы;
- в него помещается другой элемент — уже с отображением в виде строки таблицы;
- а уже в этот элемент помещаются панель навигации и основное содержание, для которых задается отображение в виде ячеек таблицы.

Элемент-таблица и элемент-строка практически всегда создаются на основе блоков.

Помимо этого, необходимо указать для секции тела страницы (тега `<body>`) величины внешних отступов, равными 0. Это устранит просветы между границами содержания страницы и краями клиентской области окна Web-обозревателя.

HTML- и CSS-код, создающий такую разметку, приведен в листинге 13.9.

Листинг 13.9

```
/* Убираем просветы между содержанием страницы и краями клиентской
области окна Web-обозревателя */
body { margin: 0px; }
/* Указываем высоту "шапки". Ширину не задаем, вследствие чего "шапка"
растянется на всю ширину клиентской области. */
header { height: 30px; }
/* Задаем отображение блока-таблицы */
div#table { display: table; }
/* Задаем отображение для блока - строки таблицы */
div#table > div { display: table-row; }
```

```
/* Задаем для панели навигации и элемента основного содержания
отображение в виде ячеек таблицы */
div#table > div > * { display: table-cell; }
/* Указываем ширину панели навигации. Это обычная практика. */
nav { width: 130px; }
/* Указываем такую ширину для элемента основного содержания, чтобы он
занял все свободное пространство, не занятое панелью навигации */
section { width: calc(100vw - 130px); }
/* Задаем высоту "поддона". Ширину не задаем, и "поддон" растянется на
всю ширину клиентской области. */
footer { height: 20px; }
. . .
<header>
  <!-- Содержимое "шапки" -->
</header>
<div id="table">
  <div>
    <nav>
      <!-- Содержимое панели павигации -->
    </nav>
    <section>
      <!-- Основное содержание -->
    </section>
  </div>
</div>
<footer>
  <!-- Содержимое "поддона" -->
</footer>
```

Страница с такой разметкой будет прокручиваться в окне Web-обозревателя целиком.

Табличная разметка с фиксированными «шапкой» и «поддоном»

Довольно эффектно выглядит *разметка с фиксированными «шапкой» и «поддоном»*. Она внешне напоминает табличную разметку, но «шапка» и «поддон» при прокрутке содержания страницы остаются на месте.

Создать такую разметку несложно:

- «шапка» и «поддон» превращаются в фиксированные элементы. Для них указывается непрозрачный фон, чтобы панель навигации и основное содержание не просвечивали сквозь них;
- для элемента-таблицы указываются величины внешних отступов сверху и внизу, равные высоте, соответственно, «шапки» и «поддона». Это нужно, чтобы панель

навигации и основное содержание не перекрывались ими в крайних положениях вертикальной полосы прокрутки страницы.

Разметка такого рода может быть создана на основе того же HTML-кода, что и обычная табличная разметка, с которой мы познакомились в предыдущем разделе главы. Поэтому мы рассмотрим только необходимый для ее формирования CSS-код, представленный в листинге 13.10.

Листинг 13.10

```
/* Убираем просветы между содержанием страницы и краями клиентской
области окна Web-обозревателя */
body { margin: 0px; }
/* Превращаем "шапку" и "поддон" в фиксированные элементы, задаем часть
параметров их местоположения и белый непрозрачный цвет фона */
header, footer { position: fixed;
                 left: 0px;
                 right: 0px;
                 background-color: white; }
/* Указываем высоту и оставшуюся часть параметров местоположения
"шапки" */
header { height: 30px;
         top: 0px; }
/* Задаем отображение блока-таблицы и устанавливаем для нее внешние
отступы сверху и снизу */
div#table { display: table;
            margin-top: 30px;
            margin-bottom: 20px; }
/* Задаем отображение для блока - строки таблицы */
div#table > div { display: table-row; }
/* Задаем для панели навигации и элемента основного содержания
отображение в виде ячеек таблицы */
div#table > div > * { display: table-cell; }
/* Указываем ширину панели навигации */
nav { width: 130px; }
/* Указываем такую ширину для элемента основного содержания, чтобы он
занял все свободное пространство, не занятое панелью навигации */
section { width: calc(100vw - 130px); }
/* Задаем высоту и часть параметров местоположения "поддона" */
footer { height: 20px;
         bottom: 0px; }
```

Табличная разметка с фиксированными «шапкой» и «поддоном», создаваемая кодом из листинга 13.10, показана на рис. 13.13. Для наглядности к элементам, составляющим разметку, были добавлены рамки, а содержание страницы было прокручено вверх.

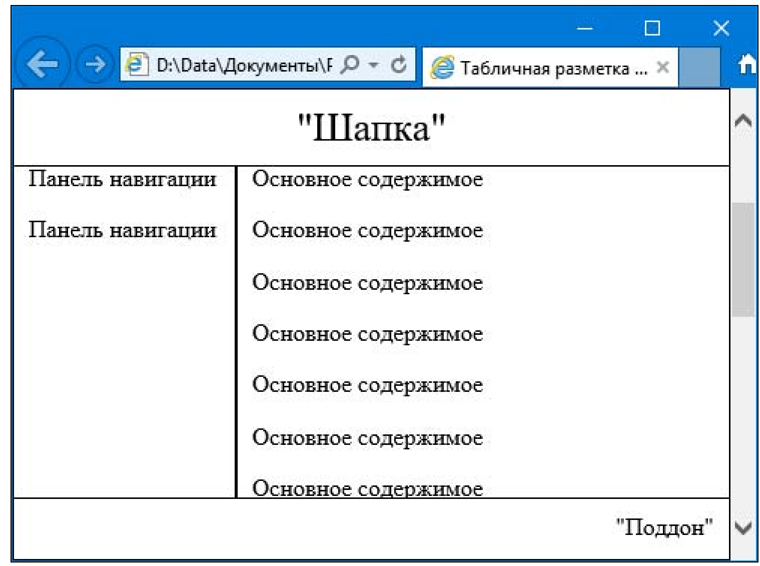


Рис. 13.13. Табличная разметка с фиксированными «шапкой» и «поддоном»

Рамочная разметка

Есть еще одна разновидность разметки, в которой прокручивается не все содержание страницы, а только содержимое некоторых элементов, а именно: основного содержания и, возможно, панели навигации. Такая разметка носит название *рамочной*.

Принципы создания рамочной разметки таковы:

- все элементы, составляющие разметку, превращаются в фиксированные;
- элементы, содержимое которых должно прокручиваться, превращаются в элементы с прокруткой (см. *главу 12*).

В рамочной разметке нет необходимости создавать дополнительные блоки, как в обычной табличной разметке, описанной ранее, а именно элемент-таблицу и элемент-строку. Достаточно создать лишь элементы, собственно составляющие эту разметку. Также нет необходимости убирать просвет между содержанием страницы и краями клиентской области окна Web-обозревателя.

В листинге 13.11 показан HTML- и CSS-код, создающий такую разметку.

Листинг 13.11

```
/* Превращаем все элементы, составляющие разметку, в фиксированные */
header, nav, section, footer { position: fixed; }
/* Указываем часть параметров местоположения "шапки" и "поддона" */
header, footer { left: 0px;
                 right: 0px; }
```

```
/* Указываем для "шапки" остальные параметры местоположения и высоту */
header { top: 0px;
         height: 30px; }
/* Задаем для панели навигации и элемента основного содержания часть
параметров местоположения и прокрутку содержимого по вертикали */
nav, section { top: 30px;
               bottom: 20px;
               overflow-y: auto; }
/* Задаем для панели навигации и элемента основного содержания остальные
параметры местоположения и ширину */
nav { left: 0px;
      width: 150px; }
section { right: 0px;
          width: calc(100vw - 150px); }
/* Осталось указать для "поддона" остальные параметры местоположения и
высоту */
footer { bottom: 0px;
         height: 20px; }
. . .
<header>
  <!-- Содержимое "шапки" -->
</header>
<nav>
  <!-- Содержимое панели павигации -->
</nav>
<section>
  <!-- Основное содержание -->
</section>
<footer>
  <!-- Содержимое "поддона" -->
</footer>
```

Такая разметка показана на рис. 13.14. Для наглядности к элементам, составляющим разметку, были добавлены рамки, а панель навигации и основное содержание были прокручены вверх.

Вероятно, для формирования рамочной разметки требуется самый лаконичный HTML- и CSS-код. Впрочем, все разновидности разметки из рассмотренных в этом разделе главы имеют определенные достоинства и недостатки, которые не являются определяющими, так что выбор разметки для создания сайта — всего лишь вопрос предпочтений Web-верстальщика.

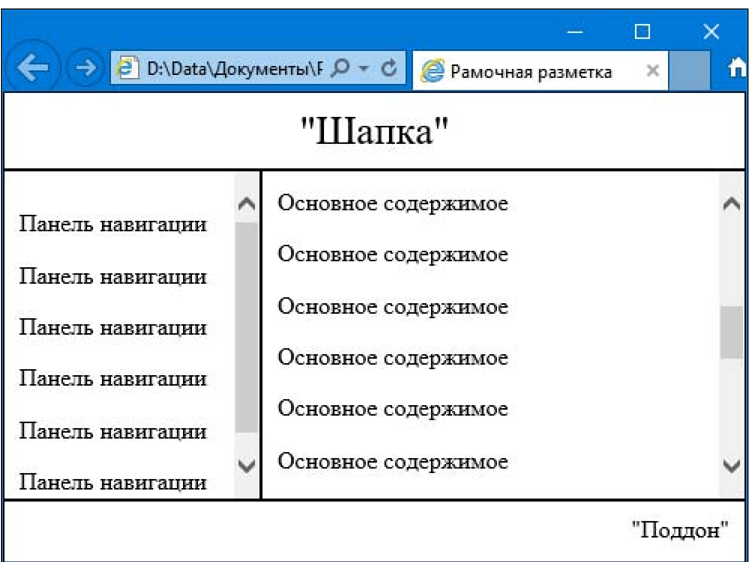
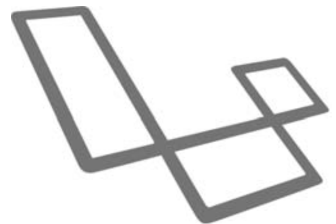


Рис. 13.14. Рамочная разметка



ГЛАВА 14

Специальные эффекты CSS 3

CSS 3 принес много нового: градиентные фоны, тени у текста и блочных элементов, гибкую и многоколоночную верстку и др. А еще преобразования элементов страницы, двухмерные и трехмерные, и анимацию. Им и будет посвящена эта глава.

Преобразования

Преобразованиями в терминологии CSS 3 называются всякого рода смещения элементов страниц относительно их изначального местоположения, наклоны, повороты и масштабирование.

Преобразования можно выполнять как в плоскости страницы, так и в воображаемом трехмерном пространстве. Преобразования первого рода носят название *двухмерных*, а преобразования второго рода — *трехмерных*.

Как задаются преобразования и их параметры?

Для указания, какое, собственно, преобразование следует применить к элементу страницы, и его параметров CSS 3 предусматривает атрибут стиля `transform`. В качестве его значения указывается функция CSS 3, задающая само выполняемое преобразование. Параметры преобразования задаются в виде аргументов этой функции:

```
transform: <функция, задающая преобразование>
```

В следующем примере мы смещаем блок `transformed` на 200 пикселов вниз:

```
div#transformed { transform: translateY(200px); }
```

В дальнейшем разговор, в основном, пойдет о функциях CSS, применяемых для указания преобразований и их параметров. Попутно мы рассмотрим еще несколько атрибутов стилей, которые могут нам пригодиться.

Двухмерные преобразования

И начнем мы с рассмотрения двухмерных преобразований CSS 3 как применяемых наиболее часто.

Смещение

Проще всего выполнить смещение элемента страницы относительно его изначального местоположения. Для этого используются три функции CSS 3, которые мы сейчас рассмотрим.

Функции `translateX` и `translateY` выполняют смещение, соответственно, по горизонтали и вертикали, или, если пользоваться математическими терминами, по горизонтальной (`x`) и вертикальной (`y`) координатным осям. Величина смещения в любой поддерживаемой CSS единице измерения указывается в качестве единственного аргумента, в круглых скобках, которые ставятся сразу же после имени функции.

Величина горизонтального смещения отсчитывается вправо, а величина вертикального смещения — вниз. Если нам потребуется сдвинуть элемент влево или вверх, мы укажем отрицательное значение соответствующей величины.

Рассмотрим пару примеров:

- сдвигаем блок `el1` на 100 пикселов вправо:

  ```
  div#el1 { transform: translateX(100px); }
  ```

- сдвигаем блок `el2` на 200 пикселов вверх:

  ```
  div#el2 { transform: translateY(-200px); }
  ```

Функция `translate` позволит нам сместить элемент сразу по обеим координатным осям. В качестве первого аргумента в ней указывается величина смещения по горизонтали, а в качестве второго — величина смещения по вертикали.

В следующем примере мы смещаем блок `el3` на 50 пикселов влево и на 200 пикселов вниз:

```
div#el3 { transform: translate(-50px, 200px) }
```

Масштабирование

Для выполнения масштабирования применяются функции CSS 3 `scaleX` и `scaleY`. Первая выполняет масштабирование вдоль горизонтальной оси координат, вторая — вдоль вертикальной.

В качестве единственного аргумента этих функций указывается относительная величина, на которую следует выполнить масштабирование элемента. Величина, бóльшая единицы, задает увеличение масштаба, а величина, меньшая единицы, — его уменьшение. Единица измерения при этом не указывается.

Точка, относительно которой элемент будет в результате масштабирования вырастать или уменьшаться, находится в его центре. (Впрочем, мы можем сместить ее в другое место — как это сделать, будет рассказано позже.)

Рассмотрим пару примеров:

- увеличиваем блок `el1` в ширину вдвое:

  ```
  div#el1 { transform: scaleX(2); }
  ```

- а блок `el2` уменьшаем вдвое в высоту:

```
div#el1 { transform: scaleY(0.5); }
```

Если нам потребуется выполнить масштабирование элемента сразу по двум координатным осям, мы воспользуемся функцией `scale`. Первым ее аргументом задается величина масштаба по горизонтали, а вторым — величина масштаба по вертикали.

Вот так мы растягиваем блок `el3` вдвое в ширину и сжимаем также вдвое, но уже в высоту:

```
div#el3 { transform: scale(2, 0.5); }
```

Наклон

Наклонить элемент страницы можно, использовав функции `skewX` и `skewY`. Первая выполняет наклон по горизонтали, вторая — по вертикали.

В качестве единственного аргумента этих функций указывается угол, на который требуется выполнить наклон. Угол этот отсчитывается против часовой стрелки, чтобы выполнить наклон по часовой стрелке, следует указать отрицательное значение угла.

Рассмотрим пару примеров:

- наклоняем блок `el1` на 25° по горизонтали против часовой стрелки:

```
div#el1 { transform: skewX(25deg); }
```

- а блок `el2` мы наклоним на 5° по вертикали по часовой стрелке:

```
div#el2 { transform: skewY(-5deg); }
```

Наклонить элемент сразу по горизонтали и вертикали удобно с помощью функции `skew`. В качестве первого аргумента она принимает угол наклона по горизонтали, а в качестве второго — угол наклона по вертикали.

Пример:

```
div#el3 { transform: skew(45deg, -60deg); }
```

Поворот

Поворот элемента на заданный угол выполняется с помощью функции `rotate`. Нужный угол задается в качестве ее единственного аргумента.

Положительное значение угла задает поворот по часовой стрелке, отрицательное — против часовой стрелки. Точка, вокруг которой поворачивается элемент, по умолчанию находится в его центре.

Рассмотрим пару примеров:

- поворачиваем блок `el1` на 45° по часовой стрелке:

```
div#el1 { transform: rotate(45deg); }
```

- поворачиваем блок `el2` на 90° против часовой стрелки. Если этот блок содержит текст, мы получим вертикальную надпись:

```
div#el2 { transform: rotate(-90deg); }
```

Трехмерные преобразования

А теперь приступим к рассмотрению трехмерных преобразований. Они применяются не так часто, как двухмерные, но во многих случаях могут выглядеть более эффектно.

Как уже отмечалось, двухмерные преобразования выполняются по горизонтальной и вертикальной координатным осям, носящим в математике наименования `x` и `y`. В случае трехмерных преобразований к ним добавляется ось `z`, направленная из точки начала координат перпендикулярно воображаемой поверхности элемента. Положительные значения координаты по этой оси отсчитываются в направлении к посетителю, отрицательные — от посетителя.

Все трехмерные преобразования элементов страницы выполняются в ортогональной проекции на поверхность их родителя.

Перспектива

Перспектива (ее также называют *глубиной перспективы*) в CSS 3 — это отсчитываемое по оси `z` расстояние между воображаемым наблюдателем (посетителем) и поверхностью родителя элемента, над которым выполняется преобразование. Фактически перспектива добавляет трехмерному изображению «глубину», благодаря чему мы получим на странице тот эффект, которого собираемся добиться, применяя трехмерное преобразование.

По умолчанию перспектива равна нулю, т. е. фактически отсутствует. Вследствие чего при выполнении трехмерных преобразований мы получим не тот результат, на который рассчитываем.

Следовательно, сначала нам нужно указать перспективу. Сделать это можно с помощью атрибута стиля `perspective`:

```
perspective: <значение>|none
```

Значение `none` убирает перспективу, делая ее равной нулю. Это, кстати, как и указывалось ранее, значение по умолчанию.

Запомним, что этот атрибут стиля и, соответственно, перспектива указывается для родителя элемента, к которому применяется преобразование, но не для самого этого элемента.

Вот пример указания перспективы — мы задаем перспективу в 300 пикселов для всех элементов, находящихся на странице:

```
body { perspective: 300px; }
```

Указание трехмерных преобразований

В случае трехмерных преобразований для смещения, масштабирования и наклона элементов по горизонтальной и вертикальной координатным осям мы можем использовать те же функции, что применяются для выполнения двухмерных преобразований. Так что здесь нам остается лишь изучить функции, специфичные лишь для собственно трехмерных преобразований, — в основном, они выполняют преобразования по координатной оси z.

Все поддерживаемые CSS функции, задаваемые ими трехмерные преобразования и примеры их использования приведены в списке далее.

- translateZ — смещение элемента по оси z. Единственный аргумент задает величину смещения, положительные значения координат отсчитываются в направлении к посетителю, отрицательные — от посетителя. Примеры:

  ```
  div#el1 { transform: translateZ(50px); }
  div#el2 { transform: translateZ(-1cm); }
  ```

- translate3d — смещение элемента одновременно по всем координатным осям. Принимает три аргумента, задающих, соответственно, смещения по осям x, y и z. Пример:

  ```
  div#el { transform: translate3d(100px, -200px, 1cm); }
  ```

- scaleZ — масштабирование по оси z. Единственный аргумент задает масштаб элемента. Пример:

  ```
  div#el { transform: scaleZ(1.5) }
  ```

  Масштабировать элемент по оси z имеет смысл только в том случае, если перед масштабированием к нему было применено преобразование поворота вокруг оси x или y. (Как задавать для элемента сразу несколько преобразований, мы узнаем позже.) В противном случае визуально этот элемент никак не изменится.

- scale3d — масштабирование сразу по всем трем координатным осям. Принимает три аргумента, задающих, соответственно, величины масштаба по осям x, y и z. Пример:

  ```
  div#el { transform: scale3d(1.5, 0.5, 2); }
  ```

- rotateX, rotateY и rotateZ — поворот элемента по координатным осям x, y и z соответственно. Единственный аргумент задает угол поворота. Примеры:

  ```
  div#el1 { transform: rotateX(45deg); }
  div#el2 { transform: rotateY(-15deg); }
  div#el3 { transform: rotateZ(0.5turn); }
  ```

Точка зрения и ее местоположение

Мы уже знаем, что перед выполнением трехмерных преобразований необходимо указать перспективу, в противном случае мы можем вообще не получить никакого видимого результата. Также мы знаем, что перспектива в CSS 3 — это расстояние

между наблюдателем и плоскостью родителя элемента, над которым выполняется преобразование.

Но где же находится этот воображаемый наблюдатель? В особой точке, носящей название *точки зрения*. Эту точку Web-обозреватель и имеет в виду, когда рассчитывает проекцию преобразуемого элемента на двумерную плоскость его родителя.

По умолчанию точка зрения находится в центре родителя элемента, к которому применяется преобразование, и отстоит от него на величину перспективы по оси `z`. И, что очень важно, всегда остается в этом месте вне зависимости от того, какое преобразование мы применяем к элементам — потомкам этого родителя.

Но в случае возникновения необходимости мы можем сместить точку зрения в другое место родителя, тем самым заставив воображаемого наблюдателя «взглянуть» на подвергаемый преобразованиям элемент с другой стороны. Это можно сделать с помощью атрибута стиля `perspective-origin`:

```
perspective-origin: <горизонтальная координата>
[<вертикальная координата>]
```

Как видим, здесь указываются лишь горизонтальная и вертикальная координаты проекции точки зрения на родитель элемента, подвергаемого преобразованиям. Координата точки зрения по оси `z` — это фактически перспектива, задаваемая атрибутом стиля `perspective`.

Параметр *`горизонтальная координата`* можно указать в виде числа с применением любой единицы измерения CSS или одного из предопределенных значений: `left` (левая граница родителя), `center` (центр родителя) или `right` (правая граница). *`Вертикальная координата`* также задается либо числом, либо одним из предопределенных значений: `top` (верхняя граница родителя), `center` (центр родителя) и `bottom` (нижняя граница). Если *`вертикальная координата`* не указана, точка зрения будет расположена по центру родителя (словно было задано значение `center`).

Значение атрибута стиля `perspective-origin` по умолчанию — `center center` (т. е., как говорилось ранее, проекция точки зрения находится в центре родителя).

Атрибут стиля `perspective-origin`, как и атрибут стиля `perspective`, указывается для родителя элементов, к которым будут применены трехмерные преобразования.

В листинге 14.1 приведен пример CSS-кода, задающего перспективу, местоположение точки зрения и трехмерное преобразование, которое будет выполнено с учетом нового расположения точки зрения.

Листинг 14.1

```
body             { perspective: 300px;
                   perspective-origin: right top; }
div#transformed: { transform: rotateY(175deg); }
```

Скрытие обратной стороны элемента

Давайте рассмотрим пример кода, представленный в листинге 14.2.

Листинг 14.2

```
body             { perspective: 300px; }
div#transformed { font-size: 36pt;
                   width: 150px;
                   background-color: black;
                   color: white;
                   transform: rotateY(175deg); }
. . .
<div id="transformed">Блок 1</div>
```

Он создает блок с именем `transformed` и поворачивает его по вертикальной оси на 175° по часовой стрелке. В результате блок окажется, если так можно сказать, перевернутым, и мы увидим его «обратную» сторону (рис. 14.1).

Рис. 14.1. Перевернутый элемент страницы с видимой «обратной» стороной

В некоторых случаях, например, при создании средствами преобразований CSS 3 трехмерных фигур, это может быть неприемлемо. Поэтому нам может потребоваться скрыть «обратную» сторону таких вот перевернутых элементов.

Для этого мы воспользуемся атрибутом стиля `backface-visibility`:

```
backface-visibility: visible|hidden
```

Значение `visible` делает «обратную» сторону элемента видимой — это поведение по умолчанию. Значение `hidden`, напротив, полностью скрывает элемент, если он оказался повернутым к посетителю своей «обратной» стороной.

Атрибут стиля `backface-visibility` также указывается непосредственно для элемента, к которому применяются трехмерные преобразования.

Если мы добавим в код из листинга 14.2 стиль:

```
div#transformed { backface-visibility: hidden; }
```

то Web-обозреватель вообще не выведет на экран блок с именем `transformed`. Что и понятно — ведь этот блок оказался повернутым к наблюдателю своей «обратной» стороной.

Режим проецирования элементов-потомков

По умолчанию потомки элемента, к которому были применены трехмерные преобразования, отображаются в его плоскости. Если мы применим трехмерные преобразования и к ним, они так и останутся проекциями на плоскость своего родителя.

Рассмотрим пример кода, представленный в листинге 14.3.

Листинг 14.3

```
body     { perspective: 300px; }
div#cont { width: 300px;
           height: 300px;
           margin-left: 100px;
           border: black 1px solid;
           transform: rotateX(45deg); }
div#ch   { width: 150px;
           height: 150px;
           background-color: black;
           transform: translate3d(50px, 50px, 100px); }
. . .
<div id="cont">
  <div id="ch"></div>
</div>
```

Здесь мы создали два вложенных друг в друга блока и задали для обоих трехмерные преобразования: блок-родитель повернули, а блок-потомок сместили, в том числе и по оси z, чтобы приподнять его над родителем.

Результат этих преобразований, который покажет нам Web-обозреватель, представлен на рис. 14.2. Видно, что блок-потомок так и находится в плоскости блока-родителя.

Однако мы можем сделать так, чтобы при применении к ним трехмерных преобразований потомки вышли из плоскости родителя и отображались в виде проекции непосредственно на странице. Это можно использовать для создания забавных эффектов.

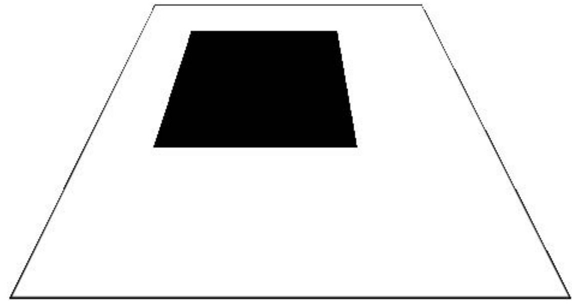


Рис. 14.2. Два вложенных друг в друга блока: блок-потомок находится в плоскости блока-родителя

Режим проецирования элементов-потомков можно указать с помощью атрибута стиля `transform-style`:

```
transform-style: flat|preserve-3d
```

- `flat` — потомки выводятся в виде проекции на поверхность родителя (поведение по умолчанию).
- `preserve-3d` — потомки существуют в воображаемом трехмерном пространстве отдельно от родителя и отображаются как проекции непосредственно на поверхности страницы.

Если мы добавим к приведенному в листинге 14.3 CSS-коду стиль:

```
div#cont { transform-style: preserve-3d; }
```

то Web-обозреватель выведет нам результат, показанный на рис. 14.3. Видно, что блок-потомок вышел за пределы блока-родителя и существует как бы отдельно от него.

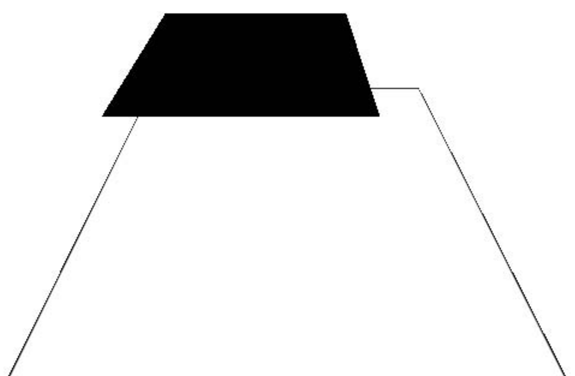


Рис. 14.3. Два вложенных друг в друга блока: блок-потомок вышел из плоскости блока-родителя

Позиционирование точки начала координат

Ранее говорилось, что по умолчанию точка начала координат располагается в центре элемента страницы, к которому применяются преобразования, но у нас есть возможность установить точку начала координат в любое место элемента. Осталось выяснить, как это делается.

Для указания месторасположения точки начала координат CSS 3 предусматривает атрибут стиля `transform-origin`:

```
transform-origin: <горизонтальная координата> <вертикальная координата>
[<координата по оси z>]
```

`Горизонтальная` и *`вертикальная координаты`* задаются так же, как у атрибута стиля `perspective-origin`. *`Координата по оси z`* указывается лишь для трехмерных преобразований. Значение по умолчанию — `center center 0px`.

Указав местоположение точки начала координат в абсолютных единицах измерения, мы можем даже вынести эту точку за пределы элемента. Иногда такой прием может быть полезным.

В следующем примере мы смещаем точку начала координат в левый верхний угол блока `el` и поворачиваем его на 45° по часовой стрелке. В результате блок будет повернут вокруг своего верхнего левого угла:

```
div#el { transform: rotate(45deg);
         transform-origin: left top; }
```

Сложные преобразования

Что ж, как сместить, отмасштабировать, наклонить или повернуть элемент страницы, мы выяснили. Но что, если нам потребуется, скажем, одновременно сместить и повернуть его, т. е. выполнить *сложное преобразование*?

Для этого достаточно все функции, указывающие отдельные преобразования, записать в значении атрибута стиля `transform` друг за другом, разделив их пробелами.

Рассмотрим несколько примеров:

- смещаем блок `el1` на 200 пикселов вниз и увеличиваем его ширину вдвое:

  ```
  div#el1 { transform: translateY(200px) scaleX(2); }
  ```

- наклоняем блок `el2` на 15° против часовой стрелки и поворачиваем на 45° также против часовой стрелки:

  ```
  div#el2 { transform: skewX(15deg) rotate(-45deg); }
  ```

- приближаем блок `el3` на 50 пикселов по направлению к посетителю, поворачиваем вокруг вертикальной оси на 45° по часовой стрелке и уменьшаем его ширину вдвое:

  ```
  div#el3 { transform: translateZ(50px) rotateY(45deg) scaleX(0.5); }
  ```

Анимация

Ранее, до появления CSS 3, для создания на страницах анимации, даже относительно простой, требовалось писать весьма сложные Web-сценарии. Но сейчас задача Web-аниматоров существенно упростилась — чтобы заставить элемент страницы менять местоположение, размер или цвет, достаточно нескольких строчек CSS-кода.

Трансформационная анимация

Проще всего создать так называемую *трансформационную анимацию*. Такая анимация формируется путем изменения значения какого-либо атрибута стиля в течение указанного нами времени и по указанному нами закону.

При создании CSS-анимации часто пользуются той же терминологией, что применяется при задании градиентных фонов (см. *главу 11*). Трансформационная анима-

ция определяется двумя ключевыми точками: первая находится в ее начальной точке и задает начальное состояние анимации, а вторая — в конечной точке и задает конечное ее состояние.

Начальное состояние анимации задается стилем, который привязывается к анимируемому элементу страницы изначально. Конечное же состояние записывается в другом стиле, который привязывается к элементу в тот момент, когда анимация должна начаться, и который, собственно, и запускает ее воспроизведение. Привязать второй стиль можно как в Web-сценарии, так и декларативно, например, применив в его селекторе псевдокласс `:hover`, указывающий на элемент, на который наводится курсор мыши. (О псевдоклассах и вообще селекторах стилей рассказывалось в *главе 8*.)

В следующем примере анимация будет запущена, как только на элемент `animated` будет наведен курсор мыши:

```
#animated {
  /* Задаем начальное состояние анимации */
}
#animated:hover {
  /* Задаем конечное состояние анимации */
}
```

Простейшая анимация

Для создания трансформационной анимации применяются пять атрибутов стилей, с которыми мы сейчас познакомимся. Все они указываются во втором стиле, задающем конечное состояние анимации.

Самый важный параметр, который мы должны указать, — это продолжительность анимации. Если мы его не зададим, элемент вообще не будет анимирован.

Продолжительность анимации задается с помощью атрибута стиля `transition-duration`:

```
transition-duration: <время>
```

Значение по умолчанию — 0, т. е. отсутствие анимации.

В листинге 14.4 приведен CSS-код, создающий анимацию, которая запустится при наведении на элемент `animated` курсора мыши и в течение трех секунд плавно изменит цвет текста с красного на зеленый.

Листинг 14.4

```
#animated       { color: #ff0000; }
#animated:hover { color: #00ff00;
                  transition-duration: 3s; }
```

Атрибут стиля `transition-property` позволяет задать атрибуты стилей, значения которых будут меняться для создания анимации:

```
transition-property: <имя атрибута стиля>|all|none
```

`Имя атрибута стиля` указывается без кавычек. Значение `all` указывает, что в анимацию будут вовлечены все атрибуты стилей, значения которых изменились (поведение по умолчанию). Значение `none` вообще отключает анимацию.

> **ВНИМАНИЕ!**
>
> Значения атрибутов стилей, которые будут вовлечены в анимацию, должны быть указаны в абсолютных единицах измерения: пикселах, пунктах, миллиметрах и т. п. Если указать их в относительных единицах измерения, например, в процентах, анимация выполнена не будет.

Код, приведенный в листинге 14.5, создает анимацию, в которую будет вовлечен лишь цвет текста, но не цвет фона. В результате цвет текста будет меняться плавно, а цвет фона, напротив, изменится скачкообразно.

Листинг 14.5

```
#animated       { color: #ff0000;
                  background-color: #ffffff; }
#animated:hover { color: #00ff00;
                  background-color: #000000;
                  transition-property: color;
                  transition-duration: 3s; }
```

Атрибут стиля `transition-delay` задает задержку перед началом анимации:

```
transition-delay: <время>
```

Значение по умолчанию — 0, т. е. анимация начинается без всякой задержки.

Атрибут стиля `transition-timing-function` устанавливает закон, по которому будет изменяться значение анимируемого атрибута стиля:

```
transition-timing-function: ease|linear|ease-in|ease-out|ease-in-out|
step-start|step-end|steps(<количество шагов анимации>[, start|end])|
cubic-bezier(<горизонтальная координата первой опорной точки>,
<вертикальная координата первой опорной точки>,
<горизонтальная координата второй опорной точки>,
<вертикальная координата второй опорной точки>)
```

Доступных значений здесь довольно много:

- `ease` — в начале скорость анимации слегка увеличивается, а в конце слегка уменьшается (поведение по умолчанию);
- `linear` — анимация выполняется с постоянной скоростью (линейный закон);
- `ease-in` — в начале скорость анимации заметно увеличивается, после чего анимация выполняется с постоянной скоростью;
- `ease-out` — анимация выполняется с постоянной скоростью, а в конце ее скорость заметно уменьшается;
- `ease-in-out` — в начале скорость анимации заметно увеличивается, а в конце заметно уменьшается;

- `cubic-bezier` — закон, подчиняющийся кубической кривой Безье. Значения координат ее опорных точек должны быть заданы в диапазоне между 0 и 1, где 0 обозначает начало анимации, а 1 — ее конец. Однако для всех атрибутов стиля, за исключением `color` и `opacity`, вертикальная координата может выходить за эти диапазоны, что может быть использовано для достижения эффекта «эластичной» анимации;
- `steps` — значение анимируемого атрибута стиля изменяется не плавно, а скачкообразно: в начале или конце одного из шагов анимации, количество которых мы задали. *Количество шагов анимации* задается в виде числа без указания единицы измерения. Если вторым значением указано `start`, изменение значения анимируемого атрибута стиля выполняется в начале каждого шага, если `end` — в его конце (это значение по умолчанию);
- `step-start` — анимируемый атрибут стиля сразу получит свое конечное значение, которое в дальнейшем не изменится (то же самое, что `steps(1, start)`);
- `step-end` — анимируемый атрибут стиля будет иметь свое начальное значение, которое в конце анимации мгновенно изменится на конечное (то же самое, что `steps(1, end)`);

Код, приведенный в листинге 14.6, создает анимацию с полусекундной задержкой перед началом ее воспроизведения и законом, согласно которому скорость анимации в начале резко увеличится, после чего останется постоянной.

Листинг 14.6

```
#animated       { color: #ff0000;
                  background-color: #ffffff; }
#animated:hover { color: #00ff00;
                  background-color: #000000;
                  transition-property: color;
                  transition-duration: 3s;
                  transition-delay: 0.5s;
                  transition-timing-function: ease-in; }
```

Еще пара примеров:

- задаем закон анимации, подчиняющийся кубической кривой Безье:

```
#animated:hover {
transition-timing-function: cubic-bezier(0.1, 0.5, 0.9, 0.5); }
```

- а процесс этой анимации будет разбит на пять шагов, и значение анимируемого атрибута стиля будет скачкообразно меняться в начале каждого шага:

```
#animated:hover { transition-timing-function: steps(5, start); }
```

Осталось рассмотреть лишь атрибут стиля `transition`, задающий сразу все параметры анимации:

```
transition: [<анимируемый атрибут стиля>] <продолжительность>
[<закон анимации>] [<задержка>]
```

Пример:

```
#animated:hover { transition: color 3s ease-in 0.5s; }
```

Обратная анимация

Мы научились создавать анимацию, запускающуюся, как только на элемент страницы наводится курсор мыши. Если мы уберем курсор с анимируемого элемента, он тотчас мгновенно вернется в свое изначальное состояние.

А можно ли сделать так, чтобы анимируемый элемент возвращался в изначальное состояние плавно, также с применением анимации, т. е. создать для него *обратную анимацию*? Разумеется, можно.

Давайте подумаем. В случае обычной, прямой анимации ее начальное состояние задает первый стиль, а конечное — второй. Тогда для создания обратной анимации начальное состояние задаст второй стиль, а конечное — первый. И нам просто нужно поместить описание обратной анимации в первый стиль.

Код, приведенный в листинге 14.7, показывает стили, задающие параметры прямой и обратной анимации.

Листинг 14.7

```
#animated       { color: #ff0000;
                  transition: color 1s; }
#animated:hover { color: #00ff00;
                  transition: color 3s; }
```

Теперь при уводе курсора мыши с анимируемого элемента цвет его текста плавно вернется к изначальному значению в течение одной секунды.

Сложная анимация

А что если нам понадобится изменять в процессе анимации значения сразу нескольких атрибутов стилей? Тогда мы можем воспользоваться возможностями CSS по созданию сложной анимации.

Сначала мы укажем все анимируемые атрибуты стилей в атрибуте `transition-property`, разделив их запятыми:

```
transition-property: color, background-color, font-size, width, height;
```

Потом мы укажем параметры анимации для всех этих атрибутов стилей, приведя их в атрибутах `transition-duration`, `transition-delay` и `transition-timing-function` также через запятую:

```
transition-duration: 3s, 2s, 3s, 1s, 1.5s;
```

Первое значение параметра будет относиться к первому анимируемому атрибуту стиля, второе — ко второму и т. д. Так, в нашем случае значение цвета текста будет

изменяться в течение трех секунд, значение цвета фона — в течение двух секунд, а значение размера шрифта — также в течение трех секунд.

Если в каком-либо из атрибутов стиля `transition-duration`, `transition-delay` или `transition-timing-function` указано меньше значений, чем атрибутов в `transition-property`, набор указанных значений будет повторяться столько раз, чтобы покрыть все указанные анимируемые атрибуты стилей.

Пример:

```
transition-property: color, background-color, font-size, width, height;
transition-duration: 3s, 2s;
```

В этом случае:

- значение атрибута стиля `color` будет изменяться в течение трех секунд;
- значение атрибута стиля `background-color` — в течение двух секунд;
- `font-size` — трех секунд;
- `width` — двух;
- `height` — трех.

Если же задать в атрибуте стиля `transition-duration`, `transition-delay` или `transition-timing-function` большее количество значений, лишние значения будут проигнорированы.

В листинге 14.8 представлен CSS-код, задающий анимацию для цвета текста и цвета фона. Для цвета фона указана полусекундная задержка анимации, и для обоих анимируемых параметров задан закон анимации `easy-in`.

Листинг 14.8

```
#animated       { color: #ff0000;
                  background-color: #ffffff; }
#animated:hover { color: #00ff00;
                  background-color: #000000;
                  transition-property: color, background-color;
                  transition-duration: 3s, 2s;
                  transition-delay: 0s, 0.5s;
                  transition-timing-function: easy-in; }
```

Покадровая анимация

Трансформационная анимация позволит нам, скажем, сделать так, чтобы элемент страницы двигался по прямой линии. Для этого достаточно указать начальную и конечную координаты начальной и конечной точек его траектории.

Но что делать, если нам нужно заставить элемент двигаться по ломаной линии? Создать покадровую анимацию.

Покадровая анимация, если вновь прибегнуть к терминологии градиентных фонов CSS 3, включает в свой состав несколько ключевых точек. Первая и последняя на-

ходятся в начальной и конечной точках анимации и задают ее начальное и конечное состояния — этим подобная анимация не отличается от рассмотренной нами ранее трансформационной анимации.

А все прочие ключевые точки располагаются между крайними точками и задают промежуточные состояния анимации. В нашем примере такими промежуточными состояниями станут точки излома траектории, по которой движется анимируемый элемент.

Состояния анимации

Первый наш шаг — описать все состояния такой анимации, сформировав так называемый *набор состояний*. Для этого применяется директива CSS 3 под названием `@keyframes`:

```
@keyframes <имя набора состояний> {
  <список состояний>
}
```

Имя набора состояний указывается без кавычек и должно быть уникальным в пределах страницы. В нем допускается использовать буквы латиницы, цифры, дефисы и знаки подчеркивания, причем начинаться оно должно с буквы.

А *список состояний* представляется в следующем формате:

```
<момент времени для описываемого состояния> {
  <параметры анимируемого элемента для описываемого состояния>
}
```

Момент времени для описываемого состояния указывается в процентах от общей продолжительности анимации (как ее задать, мы узнаем позже). Так, значение `0%` задает начало анимации, `100%` — ее конец, а `50%` — ее середину. Вместо `0%` можно использовать предопределенное значение `from`, а вместо `100%` — `to`.

Параметры анимируемого элемента для описываемого состояния записываются в виде атрибутов стилей CSS, задающих эти параметры.

Листинг 14.9 показывает код, создающий покадровую анимацию с именем `sample` из пяти состояний. Она заставит элемент страницы двигаться по траектории, имеющей вид квадрата.

Листинг 14.9

```
@keyframes sample {
  from { left: 100px; top: 50px; }
  25%  { left: 200px; top: 50px; }
  50%  { left: 200px; top: 150px; }
  75%  { left: 100px; top: 150px; }
  to   { left: 100px; top: 50px; }
}
```

Параметры анимации

Создав набор состояний анимации, мы можем задать для нее параметры. В первую очередь, это имя набора состояний и продолжительность анимации.

Имя набора состояний указывает атрибут стиля `animation-name`:

```
animation-name: <имя набора состояний>|none
```

Имя набора состояний записывается без кавычек. Значение `none` отключает любую анимацию элемента (поведение по умолчанию).

Продолжительность анимации задается атрибутом стиля `animation-duration` в том же формате, что применяется в знакомом нам атрибуте стиля `transition-duration`.

Код из листинга 14.10 сделает свободно позиционируемый элемент анимируемым при наведении курсора мыши, для чего использует созданный ранее набор состояний `sample`.

Листинг 14.10

```
#animated       { position: absolute;
                  left: 100px;
                  top: 50px;
                  width: 50px;
                  height: 50px; }
#animated:hover { animation-name: sample;
                  animation-duration: 5s; }
```

Атрибуты стилей `animation-delay` и `animation-timing-function` задают, соответственно, задержку перед началом анимации и ее закон. Здесь используются форматы, знакомые нам по атрибутам стилей `transition-delay` и `transition-timing-function`.

Листинг 14.11 показывает пример применения этих атрибутов стилей.

Листинг 14.11

```
#animated:hover { animation-name: sample;
                  animation-duration: 5s;
                  animation-delay: 1s;
                  animation-timing-function: linear; }
```

Атрибут стиля `animation-iteration-count` задает количество повторений (проходов) анимации:

```
animation-iteration-count: <количество>|infinite
```

Количество указывается в виде числа без единицы измерения. Значение `infinite` задает бесконечное повторение анимации. Значение по умолчанию — 1.

Атрибут стиля `animation-direction` позволит нам указать, в каком направлении будет воспроизводиться анимация: в прямом, как мы указали в наборе состояний, в обратном или то в прямом, то в обратном:

```
animation-direction: normal|reverse|alternate|alternate-reverse
```

- `normal` — прямое направление анимации (поведение по умолчанию).
- `reverse` — обратное направление анимации.
- `alternate` — нечетные проходы анимации воспроизводятся в прямом направлении, а четные — в обратном.
- `alternate-reverse` — нечетные проходы анимации воспроизводятся в обратном направлении, а четные — в прямом.

Понятно, что значения `alternate` и `alternate-reverse` имеет смысл указывать лишь в тех случаях, если анимация повторяется более одного раза.

Код, приведенный в листинге 14.12, создает бесконечно воспроизводящуюся анимацию, нечетные проходы которой воспроизводятся в прямом направлении, а четные — в обратном.

Листинг 14.12

```
#animated:hover { animation-name: sample;
                  animation-duration: 5s;
                  animation-iteration-count: infinite;
                  animation-direction: alternate; }
```

Атрибут стиля `animation-play-state` указывает текущее состояние анимации — должна ли она продолжать воспроизводиться или приостановится:

```
animation-play-state: running|paused
```

Значение `running` указывает Web-обозревателю продолжать воспроизведение анимации (поведение по умолчанию). А значение `paused` приостанавливает ее.

В листинге 14.13 приведен CSS-код, создающий анимируемый элемент, который начинает двигаться сразу после загрузки страницы и приостанавливается, если на него навести курсор мыши.

Листинг 14.13

```
#animated       { position: absolute;
                  left: 100px;
                  top: 50px;
                  width: 50px;
                  height: 50px;
                  animation-name: sample;
                  animation-duration: 5s;
                  animation-iteration-count: infinite; }
#animated:hover { animation-play-state: paused; }
```

Атрибут стиля animation-fill-mode указывает положение, в котором должен находиться анимируемый элемент, когда анимация не воспроизводится (еще не началась или уже закончилась):

```
animation-fill-mode: none|forwards|backwards|both
```

- none — перед началом анимации и после ее окончания параметры анимируемого элемента задаются не начальным и конечным состояниями анимации, а другими стилями, то есть Web-обозреватель в это время не принимает во внимание параметры, указанные в начальном и конечном состояниях анимации (поведение по умолчанию).
- forwards — после окончания анимации параметры анимируемого элемента будут задаваться конечным состоянием анимации, если для нее указано прямое направление, и начальным, если анимация имеет обратное направление. Перед началом анимации параметры элемента задаются другими стилями, как и в случае none.
- backwards — перед началом анимации параметры анимируемого элемента будут задаваться начальным состоянием анимации, если для нее указано прямое направление, и конечным, если анимация имеет обратное направление. После окончания анимации параметры элемента задаются другими стилями, как и в случае none.
- both — перед началом анимации параметры анимируемого элемента будут задаваться начальным состоянием анимации, если для нее указано прямое направление, и конечным, если анимация имеет обратное направление. После окончания анимации параметры анимируемого элемента будут задаваться конечным состоянием анимации, если для нее указано прямое направление, и начальным, если анимация имеет обратное направление (фактически — комбинация значений forwards и backwards).

Атрибут стиля animation пригодится любителям указывать все параметры в одном месте:

```
animation: <имя набора состояний> <продолжительность> [<закон>]
[<задержка>] [<количество повторений>] [<направление>]
[<положение, когда анимация не воспроизводится>] [<текущее состояние>]
```

Пример:

```
#animated:hover { animation: sample 5s linear infinite alternate; }
```

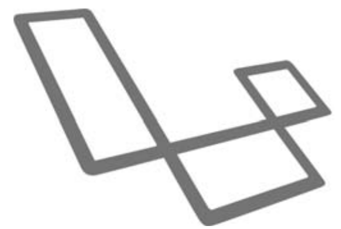
И наконец, как и в случае трансформационной анимации, мы можем указать для анимируемого элемента несколько наборов состояний и параметров к ним, создав тем самым более сложную анимацию. Пример такой анимации можно увидеть в листинге 14.14.

Листинг 14.14

```
@keyframes anim1 {
  / Набор состояний 1
}
```

```
@keyframes anim2 {
  / Набор состояний 2
}
#animated { animation-name: anim1, anim2;
            animation-direction: 3s, 6s; }
```

Здесь действуют те же правила, что мы изучили применительно к трансформационной анимации.



ГЛАВА 15

Медиазапросы. Управление выводом на печать

В последние несколько лет пользователи выходят в Интернет не только с традиционных компьютеров, но и со смартфонов, планшетов, «умных» часов, телевизоров и едва ли не с холодильников. Все эти устройства имеют различные параметры экрана и, прежде всего, разрешение и ориентацию. Следовательно, нам может понадобиться адаптировать страницы под все эти разнообразные экраны.

Кроме того, какой-то посетитель может захотеть распечатать статью, опубликованную на нашем будущем сайте. Думается, в печатной версии имеет смысл оставить лишь собственно основное содержание страницы — сам текст статьи, убрав «шапку», панель навигации и «поддон».

Как все это сделать? Использовать медиазапросы!

Использование медиазапросов

Медиазапрос можно рассматривать как набор требований к экрану устройства, записанный в определенном формате. Если экран устройства удовлетворяет этим требованиям, к странице будет применена либо отдельная таблица стилей, либо набор стилей, записанных в общей таблице, — эти стили и зададут представление, специфическое для устройства с указанными в медиазапросе параметрами.

Отсюда следует, что с помощью медиазапросов можно указать представление для устройств с различными характеристиками экрана. Так, для традиционных компьютеров можно задать для страниц полноценную разметку (о разметке говорилось в *главе 13*), включающую «шапку» и «поддон» сайта, и шрифт меньшего размера. А для старых, маломощных смартфонов можно сделать упрощенную разметку, убрав «шапку» и «поддон» и оставив лишь панель навигации и основное содержание, и увеличить размер шрифта, сделав его лучше читаемым на небольших экранах. Также для смартфонов и планшетов можно убрать стили, подсвечивающие гиперссылки при наведении на них курсора мыши, т. к. у устройств с сенсорными экранами мыши просто нет.

Медиазапросы можно применять и для создания печатных версий страниц. На печатных версиях, как говорилось ранее, имеет смысл оставить лишь основное со-

держание и, возможно, использовать для их оформления оттенки серого (все равно немалая часть принтеров, имеющихся на рынке, не может печатать в цвете).

Медиазапросы мы можем указать двумя различными способами. Поскольку каждый из них имеет свои достоинства и недостатки, рассмотрим оба.

Медиазапросы HTML

Первый способ указать медиазапрос — записать его прямо в HTML-коде страницы, создав таким образом медиазапрос HTML.

Введение в медиазапросы HTML

Медиазапрос HTML указывается в теге `<link>`, привязывающем таблицу стилей к странице. (Об этом теге мы говорили в *главе 7*.) Для этого применяется особый необязательный атрибут стиля под названием `media`, значением которого и является медиазапрос.

Встретив тег `<link>` с атрибутом `media`, Web-обозреватель проверяет, удовлетворяет ли экран устройства, на котором просматривается страница, заданным медиазапросом требованиям. Если удовлетворяет, он загрузит эту таблицу стилей и применит ее к странице, в противном случае проигнорирует.

Сама специфика медиазапросов HTML подталкивает Web-верстальщиков к размещению стилей в нескольких таблицах стилей различного назначения. Основная таблица стилей содержит стили, указывающие общее представление страницы, которое используется на всех устройствах. Эта таблица стилей привязывается к странице самой первой. За ней идут таблицы стилей, которые задают представление, специфичное для традиционных компьютеров, смартфонов и планшетов с различными параметрами экрана, печатных страниц и т. д.

Пример привязки набора таблиц стилей подобного рода с помощью медиазапроса HTML приведен в листинге 15.1.

Листинг 15.1

```
<!--Таблица стилей, задающая общее представление -->
<link href="main.css" media="all" rel="stylesheet" type="text/css">
<!--Таблица стилей, задающая представление для традиционных
компьютеров -->
<link href="pc.css" media="screen" rel="stylesheet" type="text/css">
<!--Таблица стилей, задающая представление для мобильных устройств -->
<link href="mobile.css" media="handheld" rel="stylesheet"
type="text/css">
<!--Таблица стилей, задающая представление для печати -->
<link href="print.css" media="print" rel="stylesheet" type="text/css">
```

Сам медиазапрос HTML чем-то похож на селектор стиля CSS (см. *главу 8*). Он состоит из указателей, отделенных друг от друга разделителями. Указатель обознача-

ет тип устройства или одну из характеристик его экрана. Но разделитель медиазапроса, в отличие от разделителя селектора стиля, задает способ совместного применения соседних указателей: должны ли они оба соответствовать устройству или достаточно того, чтобы устройству соответствовал лишь один из указателей.

Давайте же рассмотрим все доступные нам указатели и разделители. Их довольно много.

Указатели медиазапросов

Все поддерживаемые HTML указатели медиазапросов по типу характеристик, на которые они указывают, можно разделить на две группы.

Первая группа — это *указатели на тип устройства*. Они задают тип устройства, которому должен соответствовать медиазапрос:

- `all` — любые устройства;
- `screen` — традиционные компьютеры (имеющие большие экраны, т. е. настольные компьютеры и ноутбуки);
- `handheld` — мобильные устройства (смартфоны и планшеты);
- `tv` — «умные» телевизоры;
- `print` — печатающие устройства.

Если указатель на тип устройства отсутствует, предполагается, что медиазапрос соответствует устройствам всех типов (как если бы был задан указатель `all`).

Нужно напомнить, что примеры использования указателей на тип устройства были представлены в листинге 15.1.

Вторая группа — это *указатели на параметры устройства*. Такой указатель задает определенный параметр экрана устройства и его значение.

Указатели на параметры устройства записываются в следующем формате:

```
([<префикс>]<имя указателя>: <значение указателя>[<единица измерения>])
```

т. е. практически так же, как атрибут стиля и его значение, только берутся в круглые скобки.

Если *префикс* отсутствует, подразумевается, что указатель задает точное значение параметра. Если задан префикс `min-`, указывается минимальное значение параметра, если префикс `max-`, то максимальное. Задание префиксов поддерживает большинство указателей на параметры устройства — если какой-то из указателей их не поддерживает, мы обязательно упомянем об этом.

Для параметров размеров экрана можно использовать любую поддерживаемую CSS абсолютную *единицу измерения* CSS: пикселы, миллиметры, сантиметры и др. (см. табл. 8.1).

Перечень поддерживаемых HTML указателей на параметры устройства приведен в табл. 15.1.

Таблица 15.1. *Указатели на параметры устройств*

Указатель	Описание	Единица измерения
`width`	Ширина той части экрана устройства, что отводится под вывод страницы	Абсолютные единицы CSS (см. табл. 8.1)
`height`	Высота той части экрана устройства, что отводится под вывод страницы	
`device-width`	Ширина всего экрана устройства	
`device-height`	Высота всего экрана устройства	
`orientation`	Ориентация экрана устройства: `portrait` (портретная) или `landscape` (ландшафтная). Префиксы не поддерживаются	
`aspect-ratio`	Отношение ширины той части экрана устройства, что отводится под вывод страницы, к ее высоте. Задается в виде дроби: 4/3, 16/9 и т. п.	
`device-aspect-ratio`	Отношение ширины всего экрана устройства к его высоте. Задается в виде дроби: 4/3, 16/9 и т. п.	
`color`	Количество битов, отводимых на кодирование цвета в устройстве с цветным экраном. Для указания на устройство с монохромным экраном следует задать значение 0	
`color-index`	Количество индексированных цветов, выводимых экраном устройства. Для указания на устройство, оснащенное полноцветным экраном (не использующим таблицу цветов), следует задать значение 0	
`monochrome`	Количество битов, отводимых для вывода градаций серого в устройстве с монохромным экраном. Для указания на устройство с цветным экраном следует задать значение 0	
`resolution`	Плотность пикселов экрана устройства	`dpi` (пикселы на дюйм)
`scan`	Развертка экрана устройства: прогрессивная (`progressive`) или чересстрочная (`interlace`)	

Рассмотрим несколько примеров использования на практике указателей на параметры устройства:

- задает таблицу стилей для устройств с максимальной шириной области экрана, отведенной под вывод страницы, равной 480 пикселам:

```
<link href="styles.css" media="(max-width: 480px)" rel="stylesheet"
type="text/css">
```

- задает таблицу стилей, применяемую, если устройство расположено в ландшафтной ориентации:

```
<link href="styles.css" media="(orientation: landscape)"
rel="stylesheet" type="text/css">
```

- задает таблицу стилей для устройств с монохромными экранами:

```
<link href="styles.css" media="(color: 0)" rel="stylesheet"
type="text/css">
```

- задает таблицу стилей для экранов с плотностью пикселов не менее 400 пикселов на дюйм (т. е. для экранов типов Retina и 2K):

```
<link href="styles.css" media="(min-resolution: 400dpi)"
rel="stylesheet" type="text/css">
```

Разделители медиазапросов

Разделители медиазапросов, в отличие от их «коллег», применяемых в селекторах стилей, указывают способ совместного применения соседних указателей — это мы уже знаем. Всего таких разделителей три:

- `<запятая>` — устройство должно удовлетворять либо указателю, расположенному слева, либо указателю, расположенному справа;
- `and` — устройство должно удовлетворять и указателю, расположенному слева, и указателю, расположенному справа;
- `not` — устройство не должно удовлетворять указателю, расположенному справа.

Вот несколько примеров:

- задает таблицу стилей для мобильных устройств с максимальной шириной области экрана, отведенной под вывод страницы, равной 480 пикселам:

```
<link href="styles.css" media="handheld and (max-width: 480px)"
rel="stylesheet" type="text/css">
```

- задает таблицу стилей для печати на листах шириной 21 см (т. е. формата A4 в портретной ориентации):

```
<link href="styles.css" media="print and (device-width: 21cm)"
rel="stylesheet" type="text/css">
```

- задает таблицу стилей для традиционных компьютеров с шириной экрана от 800 до 1024 пикселов:

```
<link href="styles.css" media="screen and (min-device-width: 800px) ↳
and (max-device-width: 1024px)" rel="stylesheet" type="text/css">
```

- эта таблица стилей будет применена в странице, если последняя выводится на мобильном устройстве или на «умном» телевизоре:

```
<link href="styles.css" media="handheld, tv" rel="stylesheet"
type="text/css">
```

- а эта таблица стилей задает представление страницы для всех устройств, за исключением печатающих:

```
<link href="styles.css" media="not print" rel="stylesheet"
type="text/css">
```

Осталось сказать, что медиазапросы HTML подходят для ситуации, когда представление версий страниц, предназначенных для различных устройств, отличается настолько сильно, что удобнее создать несколько таблиц стилей, задающих разные версии представления. В противном случае лучше использовать медиазапросы CSS, речь о которых мы сейчас поведем.

Медиазапросы CSS

Медиазапрос CSS, напротив, указывается в коде таблицы стилей. Он записывается с помощью директивы CSS с «говорящим» названием `@media`:

```
@media <медиазапрос> {
  <набор стилей>
}
```

Если устройство, на котором выводится страница, удовлетворяет *медиазапросу*, указанный в фигурных скобках *набор стилей* будет применен к ней. Медиазапрос записывается с применением тех же указателей и разделителей, которые мы изучили применительно к медиазапросам HTML.

Рассмотрим пример, приведенный в листинге 15.2. Сначала там указывается для семантической «шапки» (тега `<header>`) стиль, задающий его параметры. Далее следует медиазапрос, который будет соответствовать мобильным устройствам с максимальной шириной области экрана, отводимой под вывод страницы, равной 480 пикселам, и скроет этот заголовок. В результате на мобильных устройствах с указанными параметрами экрана «шапка» выводиться не будет, а на всех остальных устройствах растянется на всю ширину страницы и будет иметь в высоту 50 пикселов.

Листинг 15.2

```
header { width: 100%;
         height: 50px; }
@media handheld and (max-width: 480px) {
  header { display: none; }
}
```

Медиазапросы CSS подойдут для тех случаев, когда версии страницы, предназначенные для различных устройств, отличаются друг от друга незначительно. Иначе целесообразнее задействовать медиазапросы HTML, разделив представление различных редакций страницы на разные таблицы стилей.

Управление выводом на печать

Раз уж зашла речь об адаптации страниц под различные устройства, в том числе и печатающие, было бы нелишне рассмотреть средства для управления их печатью. В самом деле, иногда бывает необходимо вставить перед каким-либо элементом

разрыв страницы или, наоборот, по возможности не отрывать этот элемент от его предшественников.

Атрибуты стилей `page-break-before` и `page-break-after` управляют разрывами страниц, выполняемыми, соответственно, перед или после элемента страницы:

```
page-break-before|page-break-after: auto|always|avoid|left|right
```

- `auto` — установкой разрывов страниц занимается сам Web-обозреватель (поведение по умолчанию).
- `always` — всегда вставлять разрыв страницы.
- `avoid` — по возможности исключить разрыв страницы.
- `left` — всегда вставлять разрыв страницы и формировать следующую страницу как левую.
- `right` — всегда вставлять разрыв страницы и формировать следующую страницу как правую.

Вот практический пример — перед заголовками первого уровня Web-обозреватель всегда будет выполнять разрыв страницы, а после них, по возможности, исключать его, чтобы не отрывать заголовок от следующего элемента:

```
h1 { page-break-before: always;
     page-break-after: avoid; }
```

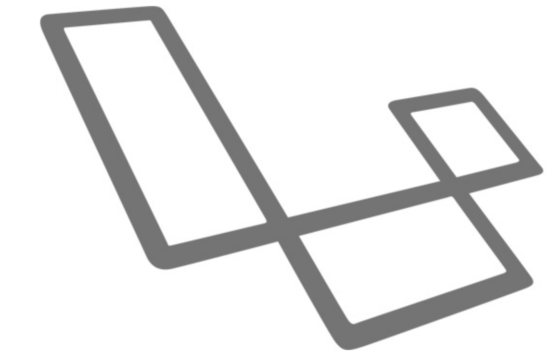
Атрибут стиля `page-break-inside` позволит нам указать Web-обозревателю не выполнять разрыв страницы внутри какого-либо элемента (опять же, по возможности):

```
page-break-inside: auto|avoid
```

Доступные для этого атрибута стиля значения уже нам знакомы.

Следующий пример определяет, что внутри абзаца, непосредственно следующего за заголовком первого уровня, разрыв страницы, по возможности, выполняться не будет:

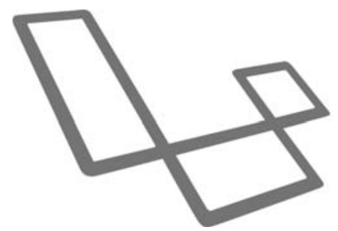
```
h1 + p { page-break-inside: avoid; }
```



I. РАЗДЕЛ 3

Поведение Web-страниц. Web-сценарии

Глава 16.	Язык программирования JavaScript
Глава 17.	Доступ к элементам страницы и управление ими
Глава 18.	Обработка событий
Глава 19.	Управление интерактивными и внедренными элементами
Глава 20.	Работа с Web-обозревателем
Глава 21.	Работа с локальными файлами. Регулярные выражения
Глава 22.	AJAX



ГЛАВА 16

Язык программирования JavaScript

К настоящему моменту мы уже знаем, как структурировать содержание Web-страницы и как указать для нее представление — с применением языков HTML и CSS соответственно. Собственно, на этом можно и остановиться, поскольку для разработки даже весьма сложных страниц этого будет вполне достаточно.

Однако если мы хотим, чтобы страница реагировала на действия посетителя, нам понадобится создать для нее поведение. Именно разработке поведения страниц и будет посвящен этот раздел.

Основные понятия JavaScript

Поведение представляет собой набор так называемых *Web-сценариев* (их еще называют *сценариями* или скриптами) — программ, которые записывают прямо в HTML-коде Web-страниц или, что предпочтительнее, в отдельных файлах. Эти программы пишутся на языке *JavaScript*. Web-обозреватель считывает JavaScript-код и последовательно выполняет записанные в нем выражения, проводя вычисления и выполняя на основе полученного результата заданные манипуляции над страницей.

Изучение JavaScript и вообще программирования удобнее начать с небольшого примера. Например, вот с такого Web-сценария:

```
var x = 4;
var y = 5;
var z = x * y;
```

Больше похоже на набор каких-то формул. Но это не формулы, а *выражения* языка JavaScript, — каждое выражение представляет собой описание одного законченного действия, выполняемого сценарием.

Разберем приведенный сценарий по выражениям. Вот первое из них:

```
var x = 4;
```

Здесь мы видим число 4. В JavaScript такие числа, а также строки и прочие величины, значения которых никогда не изменяются, называются *константами*. В самом деле, значение числа 4 всегда равно четырем.

Еще мы видим здесь латинскую букву x. Это *переменная*, которую можно рассматривать как ячейку оперативной памяти компьютера, имеющую уникальное имя и предназначенную для хранения какой-либо величины — константы или результата вычисления. Наша переменная имеет имя x.

Перед именем этой переменной поставлено слово var. Это *оператор* — команда, выполняющая определенные действия над данными. Оператор var создает переменную, чье имя расположено правее, или, как говорят программисты, выполняет *объявление переменной* (то есть var — это *оператор объявления переменной*). В JavaScript настоятельно рекомендуется объявлять переменные перед их использованием.

Осталось выяснить, что делает символ равенства (=), поставленный между переменной и константой. Это тоже оператор, помещающий значение, расположенное справа (*операнд*), в переменную, расположенную слева, — в нашем случае значение 4 в переменную x. Он носит название *оператора присваивания*.

В результате рассматриваемое нами выражение объявит новую переменную x и присвоит ей значение константы 4.

Каждое выражение JavaScript должно оканчиваться символом точки с запятой (;), обозначающим конец выражения. Его отсутствие вызывает ошибку обработки сценария.

Рассмотрим следующее выражение:

```
var y = 5;
```

Оно аналогично первому — объявляет переменную y и присваивает ей константу 5.

Третье выражение стоит несколько особняком:

```
var z = x * y;
```

Здесь мы видим все те же операторы объявления переменной и присваивания, причем последний присваивает что-то переменной z. Но что? Результат вычисления произведения значений, хранящихся в переменных x и y. Вычисление произведения выполняет оператор умножения, который в JavaScript обозначается символом звездочки (*). Это *арифметический оператор*.

В результате выполнения приведенного ранее Web-сценария в переменной z окажется произведение значений 4 и 5, то есть 20.

Вот еще один пример математического выражения, на этот раз более сложного (предполагается, что переменные x1, x2, y1 и y2 уже объявлены и хранят какие-то числовые значения):

```
var y = y1 * y2 + x1 * x2;
```

Оно вычисляется в следующем порядке:

1. Значение переменной y1 умножается на значение переменной y2.
2. Перемножаются значения переменных x1 и x2.

3. Полученные на шагах 1 и 2 произведения складываются (оператор сложения обозначается привычным нам знаком `+`).
4. Полученная сумма присваивается переменной `y`.

Но почему на шаге 2 выполняется умножение `x1` на `x2`, а не сложение произведения `y1` и `y2` с `x1`. Дело в том, что каждый оператор имеет *приоритет* — своего рода номер в очереди их выполнения. Так вот, оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому умножение всегда выполняется перед сложением.

А вот еще одно выражение (предполагается, что переменная `x` уже объявлена и хранит какое-то число):

```
x = x + 3;
```

Оно абсолютно правильно с точки зрения JavaScript, хоть и выглядит нелепым. В нем сначала выполняется сложение значения переменной `x` и числа 3, после чего результат сложения снова присваивается переменной `x`. Такие выражения встречаются в сценариях довольно часто.

Типы данных JavaScript

JavaScript может манипулировать данными, относящимися к разным типам. *Тип данных* описывает их возможные значения и набор применимых к ним операций. Давайте рассмотрим все типы данных, с которыми мы можем столкнуться.

Строковые данные (или *строки*) — это последовательности букв, цифр, пробелов, знаков препинания и прочих символов, заключенные в одинарные или двойные кавычки.

Примеры строк:

```
"JavaScript"
"1234567"
'Строковые данные — это последовательности символов.'
```

Строки могут также содержать *специальные символы*, служащие для особых целей (табл. 16.1).

Таблица 16.1. *Специальные символы, поддерживаемые JavaScript, и их коды*

Символ	Описание
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Табуляция
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратный слеш
`\x`*<код>*	Любой символ по его *коду* в кодировке Unicode

Таким образом, если нам требуется поместить в строку двойные кавычки, нужно записать ее так:

```
"\"Этот фрагмент текста\" помещен в кавычки"
```

Числовые данные (или *числа*) — это обычные числа, над которыми можно производить арифметические действия, извлекать из них квадратный корень и вычислять тригонометрические функции. Числа могут быть как целыми, так и дробными, — в последнем случае целая и дробная части разделяются точкой (не запятой!).

Примеры чисел:

```
13756
454.7873
0.5635
```

Дробные числа могут быть записаны в экспоненциальной форме:

`<мантисса>E<порядок>`.

Вот примеры заданных таким образом чисел (в скобках дано традиционное математическое представление):

`1E-5` (10^{-5})

`8.546E23` ($8{,}546 \cdot 10^{23}$)

Также имеется возможность записи целых чисел в восьмеричном и шестнадцатеричном виде. Восьмеричные числа записываются с нулем в начале (например, `047` или `-012543624`), а шестнадцатеричные — с символами `0x`, также помещенными в начало (например, `0x35F`). Отметим, что в JavaScript так можно записывать только целые числа.

К числовому типу относятся также следующие специальные значения:

- `Infinity` — математическая ∞ (бесконечность);
- `-Infinity` — математическая $-\infty$ (минус бесконечность);
- `NaN` — Not a Number, не число. Является результатом вычисления арифметических выражений с операндами разных типов — например, при попытке умножения числа на строку.

Логическая величина может принимать только два значения: «true» и «false» — «истина» и «ложь», — обозначаемые соответственно ключевыми словами `true` и `false`. (*Ключевое слово* — это слово, имеющее в языке программирования особое значение.) Логические величины используются, как правило, в условных выражениях (о них речь пойдет далее).

Величина особого типа *undefined* может принимать одно-единственное значение — `undefined`. Это значение получит только что объявленная переменная, которой еще не было присвоено какое-либо иное значение. (Об объявлении переменных мы подробнее поговорим чуть позже.)

Логические величины и величину `undefined` более подробно мы рассмотрим далее. Равно как и познакомимся с другими двумя типами данных, которые здесь еще не упоминались.

Переменные

В начале этой главы мы кое-что узнали о переменных. Сейчас настало время обсудить их детальнее.

Именование переменных

Как мы уже знаем, каждая переменная должна иметь имя, которое однозначно ее идентифицирует. Об именах переменных стоит поговорить подробнее.

Прежде всего, в имени переменной могут присутствовать только латинские буквы, цифры, символы подчеркивания (_) и доллара ($), причем первый символ имени должен быть либо буквой, либо символом подчеркивания, либо символом доллара. Например: `pageAddress`, `_link`, `$userName` — правильные имена переменных, а `678vasya` и `Имя пользователя` — неправильные.

Язык JavaScript чувствителен к регистру символов, которыми набраны имена переменных. Это значит, что `pageaddress` и `pageAddress` — разные переменные.

> ***Внимание!***
>
> Это правило также соблюдается при написании операторов, ключевых слов, с помощью которых образуются языковые конструкции, имен функций, объектов и проч. Поэтому говорят, что JavaScript является регистрозависимым языком.

Совпадение имени переменной с ключевым словом языка JavaScript не допускается.

Объявление переменных

Как говорилось ранее, перед использованием переменной в коде Web-сценария рекомендуется выполнить ее объявление. Для этого служит уже знакомый нам оператор объявления переменной `var`, после которого указывают имя переменной:

```
var x;
```

Сразу после объявления переменная получит значение `undefined`. Но, разумеется, мы можем присвоить ей другое, более полезное значение:

```
x = 1234;
```

и использовать в сценарии:

```
y = x * 2 + 10;
```

Значение переменной также можно присвоить прямо при ее объявлении:

```
var x = 1234;
```

Можно и сразу объявить несколько переменных:

```
var x, y, textColor = "black";
```

Вообще, объявлять переменные с помощью оператора `var` не обязательно. Мы можем просто присвоить переменной какое-либо значение, и JavaScript сам ее создаст.

Просто явное объявление переменных оператором `var` считается хорошим стилем программирования.

Переменная, созданная в каком-либо Web-сценарии, будет доступна во всех остальных сценариях, присутствующих в разрабатываемой странице. Об исключениях из этого правила мы поговорим позже.

Операторы

Операторов язык JavaScript поддерживает очень много — на все случаи жизни. Их можно разделить на несколько групп.

Арифметические операторы

Арифметические операторы служат для выполнения арифметических действий над числами. Все арифметические операторы, поддерживаемые JavaScript, приведены в табл. 16.2.

Таблица 16.2. *Арифметические операторы*

Оператор	Описание
`-`	Смена знака числа
`+`	Сложение
`-`	Вычитание
`*`	Умножение
`/`	Деление
`%`	Взятие остатка от деления
`++`	*Инкремент* (увеличение на единицу)
`--`	*Декремент* (уменьшение на единицу)

Примеры выражений с операторами:

```
m = -m;
```

Меняет знак числа, хранящегося в переменной `m`, на противоположный и присваивает его той же переменной.

```
t = n + 2;
b = a * 2 / 3;
```

Здесь все понятно.

```
++r;
```

Увеличивает значение, хранящееся в переменной `r`, на единицу (инкрементирует его).

```
s = ++r;
```

Увеличивает значение, хранящееся в переменной `r`, на единицу и присваивает его переменной `s`.

Операторы инкремента и декремента можно ставить как перед операндом, так и после него. Если оператор инкремента стоит перед операндом, то значение операнда сначала увеличивается на единицу, а уже потом используется в дальнейших вычислениях. Если же оператор инкремента стоит после операнда, то его значение сначала вычисляется, а уже потом увеличивается на единицу. Точно так же ведет себя оператор декремента.

Оператор объединения строк

Оператор объединения строк `+` позволяет соединить две строки в одну. Например, сценарий:

```
s1 = "Java";
s2 = "Script";
s = s1 + s2;
```

поместит в переменную `s` строку `"JavaScript"`.

Операторы присваивания

Оператор присваивания `=` нам уже знаком. Его еще называют оператором *простого присваивания*, поскольку он просто присваивает переменной новое значение:

```
a = 2;
b = c = 3;
```

Второе выражение в приведенном примере выполняет присвоение значения `3` сразу двум переменным — `b` и `c`.

JavaScript также поддерживает *операторы сложного присваивания*, позволяющие выполнять присваивание одновременно с другой операцией:

```
a = a + b;
a += b;
```

Два этих выражения эквивалентны по результату. Просто во втором указан оператор сложного присваивания `+=`.

Все операторы сложного присваивания, поддерживаемые JavaScript, и их эквиваленты приведены в табл. 16.3.

Таблица 16.3. *Операторы сложного присваивания*

Оператор	**Эквивалентное выражение**
`a += b;`	`a = a + b;`
`a -= b;`	`a = a - b;`

Таблица 16.3 *(окончание)*

Оператор	Эквивалентное выражение
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`

Операторы сравнения

Операторы сравнения сравнивают два операнда согласно определенному условию и выдают (или, как говорят программисты, *возвращают*) логическое значение. Если условие сравнения выполняется, возвращается значение `true`, если не выполняется — `false`.

Все поддерживаемые JavaScript операторы сравнения приведены в табл. 16.4.

Таблица 16.4. *Операторы сравнения*

Оператор	Описание	Оператор	Описание
<	Меньше	==	Равно
>	Больше	!=	Не равно
<=	Меньше или равно	===	Строго равно (см. далее)
>=	Больше или равно	!==	Строго не равно (см. далее)

Примеры:

```
var a1 = 2 < 3;
var a2 = -4 > 0;
var a3 = r < t;
```

Здесь переменная `a1` получит значение `true` (2 меньше 3), переменная `a2` — значение `false` (число –4 по определению не может быть больше нуля), а значение переменной `a3` будет зависеть от значений переменных `r` и `t`.

Можно сравнивать не только числа, но и строки:

```
var a = "JavaScript" != "Java";
```

Переменная `a` получит значение `true`, т. к. строки `"JavaScript"` и `"Java"` не равны.

Логические операторы

Логические операторы выполняют действия над логическими значениями. Все они приведены в табл. 16.5. А в табл. 16.6 и 16.7 показаны результаты выполнения этих операторов.

Таблица 16.5. *Логические операторы*

Оператор	Описание
`!`	НЕ (логическая инверсия)
`&&`	И (логическое умножение)
`||`	ИЛИ (логическое сложение)

Таблица 16.6. *Результаты выполнения операторов И и ИЛИ*

Операнд 1	Операнд 2	&& (И)	|| (ИЛИ)
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Таблица 16.7. *Результаты выполнения оператора НЕ*

Операнд	! (НЕ)
`true`	`false`
`false`	`true`

Основная область применения логических операторов — выражения сравнения (о них см. далее в этой главе). Приведем примеры таких выражений:

```
var a = b > 0 && c + 1 != d;
flag = !(status = 0);
```

Оператор получения типа *typeof*

Оператор получения типа `typeof` возвращает строку, описывающую тип данных операнда. Операнд, тип которого нужно узнать, помещается после этого оператора через пробел.

```
s = typeof "str";
```

В результате выполнения этого выражения в переменной `s` окажется строка `"string"`, обозначающая строковый тип.

Все значения, которые может вернуть оператор `typeof`, приведены в табл. 16.8.

Таблица 16.8. *Значения, возвращаемые оператором* `typeof`

Тип данных	Возвращаемая строка
Строковый	`"string"`
Числовой	`"number"`

Таблица 16.8 *(окончание)*

Тип данных	Возвращаемая строка
Логический	"boolean"
Объектный (см. далее)	"object"
Функциональный (см. далее)	"function"
undefined	"undefined"

Преобразование типов данных

Что получится, если сложить два числовых значения? Правильно — еще одно числовое значение. А если сложить число и строку? Трудно сказать... Тут JavaScript сталкивается с проблемой несовместимости типов данных и пытается сделать эти типы совместимыми, преобразуя один из них к другому. Сначала он пытается преобразовать строку в число и, если это удается, выполняет сложение. В случае неудачи число будет преобразовано в строку, и две полученные строки будут объединены. Например, в результате выполнения Web-сценария из листинга 16.1 значение переменной `b` при сложении с переменной `a` будет преобразовано в числовой тип, — таким образом, переменная `c` будет содержать значение 23.

Листинг 16.1

```
var a, b, c, d, e, f, g;
a = 11;
b = "12";
c = a + b;
d = "JavaScript";
e = 2;
f = d + e;
g = 2 / "Java"
```

Но т. к. значение переменной `d` нельзя преобразовать в число, значение `e` будет преобразовано в строку, и результат — значение `f` — станет равным `"JavaScript2"`.

А в переменной `g` окажется значение `NaN`, поскольку строку `"Java"` нельзя преобразовать в число, и разделить число на строку также нельзя.

Логические величины преобразуются либо в числовые, либо в строковые, в зависимости от конкретного случая. Значение `true` будет преобразовано в число 1 или строку `"true"`, а значение `false` — в 0 или `"false"`. И наоборот, число 1 будет преобразовано в значение `true`, а число 0 — в значение `false`. Также в `false` будут преобразованы значения `null` и `undefined`.

Преобразование типов будет выполняться и при вычислении результата операторов сравнения (см. табл. 16.4). Так, обычные операторы «равно» и «не равно», если

встречают операнды разных типов, сначала пытаются преобразовать их к одному типу.

Однако это не касается операторов «строго равно» и «строго не равно». Это операторы так называемого *строгого сравнения*. Они преобразования типов не выполняют, а в случае несовпадения типов операндов всегда возвращают `false`.

```
var a1 = 2 == "2";
var a2 = 2 === "2";
```

Здесь переменная `a1` получит значение `true`, т. к. в процессе вычисления строка `"2"` будет преобразована в число 2, и условие сравнения выполнится. Но переменная `a2` получит значение `false`, т. к. сравниваемые операнды принадлежат разным типам.

Приоритет операторов

Последний вопрос, который мы здесь рассмотрим, — приоритет операторов. Как мы помним, приоритет влияет на порядок, в котором выполняются операторы в выражении.

Пусть имеется следующее выражение:

```
a = b + c — 10;
```

В этом случае сначала к значению переменной `b` будет прибавлено значение `c`, а потом из суммы будет вычтено 10. Операторы этого выражения имеют одинаковый приоритет и поэтому выполняются строго слева направо.

Теперь обратимся к такому выражению:

```
a = b + c * 10;
```

Здесь сначала будет выполнено умножение значения `c` на 10, а уже потом к полученному произведению будет прибавлено значение `b`. Оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому порядок «строго слева направо» будет нарушен.

Самый низкий приоритет у операторов присваивания. Вот почему сначала вычисляется само выражение, а потом его результат присваивается переменной.

В общем, основной принцип выполнения всех операторов таков: сначала выполняются операторы с более высоким приоритетом, а уже потом — операторы с более низким. Операторы с одинаковым приоритетом выполняются в порядке их следования (слева направо).

В табл. 16.9 приведены все изученные нами операторы в порядке убывания их приоритетов.

Таблица 16.9. *Приоритет операторов (в порядке убывания)*

Операторы	Описание
`++ -- - ~ ! typeof`	Инкремент, декремент, смена знака, логическое НЕ, определение типа

Таблица 16.9 *(окончание)*

<table>
<tr><th>Операторы</th><th>Описание</th></tr>
<tr><td><code>* / %</code></td><td>Умножение, деление, взятие остатка</td></tr>
<tr><td><code>+ -</code></td><td>Сложение и объединение строк, вычитание</td></tr>
<tr><td><code>< > <= >=</code></td><td rowspan="2">Операторы сравнения</td></tr>
<tr><td><code>== != === !==</code></td></tr>
<tr><td><code>&&</code></td><td>Логическое И</td></tr>
<tr><td><code>||</code></td><td>Логическое ИЛИ</td></tr>
<tr><td><code>?</code></td><td>Условный оператор (см. далее)</td></tr>
<tr><td><code>=</code> <оператор><code>=</code></td><td>Присваивание, простое и сложное</td></tr>
</table>

Но что делать, если нам нужно нарушить обычный порядок выполнения операторов? Воспользуемся скобками. При такой записи заключенные в скобки операторы выполняются первыми:

```
a = (b + c) * 10;
```

Здесь сначала будет выполнено сложение значений переменных `b` и `c`, а потом получившаяся сумма будет умножена на 10.

Операторы, заключенные в скобки, также подчиняются приоритету. Поэтому часто используются многократно вложенные скобки:

```
a = ((b + c) * 10 - d) / 2 + 9;
```

Здесь операторы будут выполнены в такой последовательности:

1. Сложение `b` и `c`.
2. Умножение полученной суммы на 10.
3. Вычитание `d` из произведения.
4. Деление разности на 2.
5. Прибавление 9 к частному.

Если же удалить скобки:

```
a = b + c * 10 - d / 2 + 9;
```

то порядок выполнения операторов будет таким:

1. Умножение `c` и 10.
2. Деление `d` на 2.
3. Сложение `b` и произведения `c` и 10.
4. Вычитание из полученной суммы частного от деления `d` на 2.
5. Прибавление 9 к полученной разности.

Получается совсем другой результат, не так ли?

Блоки

JavaScript позволяет нам объединить несколько выражений в одно. Такое выражение называется *блочным выражением* или просто *блоком*. Составляющие его выражения заключают в фигурные скобки.

Пример:

```
{
  b = "12";
  c = a - b;
}
```

Блоки используются в составе более сложных выражений, называемых *управляющими конструкциями* и описанными далее.

Управляющие конструкции

Обычные выражения выполняются в том порядке, в котором они записаны в коде сценария. Однако существует ряд выражений, которые позволяют нарушить этот порядок с определенной целью. Это *управляющие конструкции* JavaScript.

Ветвление

Ветвление позволяет нам выполнить одно из двух входящих в него выражений в зависимости от выполнения или невыполнения какого-либо *условия*. В качестве условия используется значение логической переменной или результат вычисления логического выражения.

Для написания ветвлений используется следующий формат:

```
if (<условие>)
  <блок if>
[else
  <блок else>]
```

Если `условие` возвращает значение `true` (условие истинно), то выполняется `блок if`. Если же `условие` возвращает значение `false` (условие ложно), то выполняется `блок else` (конечно, если он присутствует). А если `блок else` отсутствует, выполняется следующее выражение сценария.

В листинге 16.2 мы сравниваем значение переменной `x` с единицей и в зависимости от результатов сравнения присваиваем переменным `a` и `b` разные значения.

Листинг 16.2

```
if (x == 1) {
  a = "Единица";
  b = 1;
```

```
} else {
  a = "Не единица";
  b = 22222;
}
```

Условие может быть довольно сложным (листинг 16.3).

Листинг 16.3

```
if (x == 1 && y > 10)
  f = 3;
else
  f = 33;
```

Здесь мы использовали сложное условие, возвращающее значение `true` в случае, если значение переменной `x` равно 1 и значение переменной `y` больше 10. Заметим также, что мы подставили одиночные выражения, т. к. фрагменты кода слишком просты, чтобы оформлять их в виде блоков.

Выражение *множественного ветвления* позволяет объединить в себе несколько обычных выражений ветвления:

```
if (<условие 1>)
  <блок if 1>
else if (<условие 2>)
  <блок if 2>
. . .
else if (<условие n>)
  <блок if n>
[else
  <блок else>]
```

Если значение *условия 1* равно `true`, будет выполнен *блок if 1*. Если же *условие 1* равно `false`, будет проверено *условие 2*, и, если оно равно `true`, будет выполнен *блок if 2*. Если и *условие 2* равно `false`, проверяется *условие 3*, и т. д. В случае, когда все имеющиеся в ветвлении *условия* равны `false`, выполнится *блок else*.

Множественное ветвление может включать произвольное количество *условий* и соответствующих им *блоков if*. *Блок else*, как и в случае простого ветвления, может быть опущен.

В листинге 16.4 показан пример множественного ветвления.

Листинг 16.4

```
if (mass > 50)
  s = "очень тяжелый";
else if (mass <= 50 && mass > 20)
  s = "тяжелый";
```

```
else if (mass <= 20 && mass > 5)
  s = "можно поднять";
else
  s = "легкий";
```

Оператор ветвления ?

Если выражение ветвления совсем простое, мы можем записать его с помощью *оператора ветвления* `?`:

```
<условие> ? <выражение "то"> : <выражение "иначе">;
```

Достоинство этого оператора в том, что он может быть частью выражения. Например:

```
f = (x == 1 && y > 10) ? 3 : 33;
```

Фактически мы записали ветвление из листинга 16.3, но в более компактном виде. Недостаток же оператора `?` в том, что его можно применять только в самых простых случаях.

Переключение

Выражение *переключения* последовательно сравнивает указанное значение с различными константами и в случае равенства с какой-либо из них выполняет соответствующий фрагмент кода. Записывается оно в следующем формате:

```
switch (<значение>) {
  case <константа 1> :
    <фрагмент кода 1>
    break;
  case <константа 2> :
    <фрагмент кода 2>
    break;
  . . .
  case <константа n> :
    <фрагмент кода n>
    break;
  [default :
    <фрагмент кода default>]
}
```

Здесь *`значение`* последовательно сравнивается с *`константой 1`*, *`константой 2`* и т. д. и, если такое сравнение прошло успешно, выполняется соответствующий фрагмент кода (*`фрагмент кода 1`*, *`фрагмент 2`* и т. д.). Если же ни одно сравнение не увенчалось успехом, выполняется *`фрагмент кода default`* (если, конечно, он присутствует).

Листинг 16.5 иллюстрирует пример выражения выбора.

Листинг 16.5

```
switch (a) {
  case 1 :
    out = "Единица";
    break;
  case 2 :
    out = "Двойка";
    break;
  case 3 :
    out = "Тройка";
    break;
  default :
    out = "Другое число";
}
```

Здесь, если переменная `a` содержит значение 1, переменная `out` получит значение `"Единица"`, если 2 — значение `"Двойка"`, а если 3 — значение `"Тройка"`. Если же переменная `a` содержит какое-то другое значение, переменная `out` получит значение `"Другое число"`.

Циклы

Циклы — это особые выражения, выполняющие один и тот же фрагмент кода раз за разом, пока остается истинным или ложным заданное условие.

Цикл со счетчиком

Цикл со счетчиком удобен, если какой-то код нужно выполнить строго определенное число раз. Вероятно, это наиболее распространенная разновидность циклов.

Цикл со счетчиком записывается так:

```
for (<выражение инициализации>; <условие>; <приращение>)
  <тело цикла>
```

Выражение инициализации выполняется самым первым и всего один раз. Оно присваивает особой переменной, называемой *счетчиком цикла*, некое начальное значение (обычно 0 или 1). Счетчик цикла подсчитывает, сколько раз было выполнено *тело цикла* — собственно код, который нужно выполнить определенное количество раз.

Следующий шаг — проверка *условия*, выражения, результатом вычисления которого станет логическая величина. Если это `true`, выполняется *тело цикла*, в противном случае цикл завершается и начинается выполнение кода, следующего за циклом.

После прохода *тела цикла* выполняется выражение *приращения*, изменяющее значение счетчика, обычно инкрементирующее его. Далее снова проверяется *условие*, выполняется *тело цикла*, *приращение* и т. д., пока *условие* не станет равно `false`.

Пример цикла со счетчиком можно увидеть в листинге 16.6.

Листинг 16.6

```
for (var i = 1; i < 11; i++) {
  a += 3;
  b = i * 2 + 1;
}
```

Этот цикл будет выполнен 10 раз. Мы присваиваем счетчику `i` начальное значение 1 и после каждого выполнения тела цикла инкрементируем его. Цикл перестанет выполняться, когда счетчик получит значение 11, и условие цикла станет ложным.

Значение счетчика цикла можно использовать в теле цикла, как это сделали мы. В нашем случае счетчик `i` будет содержать последовательно возрастающие значения от 1 до 10, которые задействуются в вычислениях.

Другой пример цикла со счетчиком показывает листинг 16.7.

Листинг 16.7

```
for (var i = 10; i > 0; i--) {
  a += 3;
  b = i * 2 + 1;
}
```

Здесь значение счетчика декрементируется. Начальное его значение равно 10. Цикл выполнится 10 раз и завершится, когда счетчик `i` будет содержать 0. При этом значения счетчика будут последовательно уменьшаться от 10 до 1.

Тело цикла можно записать в виде обычного выражения, а не блока:

```
for (var i = 2; i < 21; i += 2) b = i * 2 + 1;
```

Здесь начальное значение счетчика равно 2, а конечное — 21, но цикл выполнится, опять же, 10 раз. А все потому, что значение счетчика каждый раз увеличивается на 2, и счетчик последовательно принимает значения 2, 4, 6, ..., 20.

Цикл с предусловием

Цикл с предусловием во многом похож на цикл со счетчиком — он выполняется до тех пор, пока остается истинным условие цикла, и условие это вычисляется и проверяется перед выполнением тела цикла. Формат записи такого цикла:

```
while (<условие>)
  <тело цикла>
```

Пример:

```
while (a < 100) {
  a = a * i + 2;
  ++i;
}
```

Цикл с постусловием

А у *цикла с постусловием* условие вычисляется и проверяется не до, а после выполнения тела цикла. Отметим, что такой цикл выполнится хотя бы один раз, даже если его условие с самого начала ложно.

Формат цикла с постусловием:

```
do
  <тело цикла>
while (<условие>);
```

Листинг 16.8 показывает пример такого рода цикла.

Листинг 16.8

```
do {
  a = a * i + 2;
  ++i;
} while (a < 100);
```

Прерывание и перезапуск цикла

Иногда нужно прервать выполнение цикла. Для этого JavaScript предоставляет Web-программистам операторы `break` и `continue`.

Оператор прерывания `break` позволяет *прервать* выполнение цикла и перейти к следующему за ним выражению (листинг 16.9).

Листинг 16.9

```
while (a < 100) {
  a = a * i + 2;
  if (a > 50) break;
  ++i;
}
```

В этом примере мы прерываем выполнение цикла, если значение переменной `a` превысит 50.

Оператор перезапуска `continue` позволяет *перезапустить* цикл, т. е. оставить невыполненными все последующие выражения, входящие в тело цикла, и запустить выполнение очередного прохода цикла: проверку условия, выполнение приращения и тела и т. д. (листинг 16.10).

Листинг 16.10

```
while (a < 100) {
  i = ++i;
  if ((i > 9) && (i < 11)) continue;
  a = a * i + 2;
}
```

В этом цикле для всех значений `i` от 10 до 20 последнее выражение тела цикла выполнено не будет.

Функции

Функция — это особым образом написанный и оформленный фрагмент кода JavaScript, который можно вызвать из любого сценария, включенного в страницу (*повторно используемый код*, как его часто называют). Так что, если какой-то фрагмент кода встречается в нескольких местах нашего сценария, рекомендуется оформить его в виде функции.

Объявление функций

Прежде чем функция будет использована где-то в сценарии, ее нужно объявить. Формат объявления функции следующий:

```
function <имя функции>([<список параметров, разделенных запятыми>]) {
  <тело функции>
}
```

Имя функции должно быть уникально в пределах страницы. Для имен функций действуют те же правила, что и для имен переменных.

Функция может принимать произвольное количество *параметров*. Их можно рассматривать как переменные, доступные только внутри функции, — этим переменным будут присвоены значения аргументов, указанных при вызове функции (вызов функций мы рассмотрим позже).

Список параметров, принимаемых функцией, указывается в круглых скобках после ее имени, сами параметры отделяют друг от друга запятыми. Если функция не принимает параметров, следует указать пустые скобки — это обязательно.

Тело функции — это код, ради которого она, собственно, и создается. Оно всегда заключается в фигурные скобки.

Функция может выдавать или, как говорят программисты, *возвращать* результат. Для этого используется *оператор возврата* `return`:

```
return <переменная или выражение>;
```

Здесь *переменная* должна содержать возвращаемое значение, а *выражение* должно его вычислять.

Пример объявления функции приведен в листинге 16.11. Эта функция принимает два параметра: `a` и `b`, после чего делит `a` на `b` и возвращает частное от этого деления.

Листинг 16.11

```
function divide(a, b) {
  var c;
  c = a / b;
  return c;
}
```

Объявление этой функции можно записать компактнее:

```
function divide(a, b) {
  return a / b;
}
```

Или даже так, в одну строку:

```
function divide(a, b) { return a / b; }
```

Если при вызове функции указано количество аргументов меньшее, чем количество записанных в ее объявлении параметров, оставшиеся параметры получат значение `undefined`. Это можно использовать, чтобы создать так называемые *необязательные параметры*, для которых можно не указывать аргументы — в этом случае они получат некоторое значение по умолчанию. Принцип создания таких параметров иллюстрирует листинг 16.12.

Листинг 16.12

```
function divide(a, b) {
  if (b === undefined) b = 2;
  return a / b;
}
```

Здесь в теле функции мы проверяем, хранит ли второй параметр (`b`) функции значение `undefined`, и, если это так, присваиваем ему значение по умолчанию (2).

Функции и переменные. Локальные переменные

В листинге 16.11 показано объявление функции, в теле которой объявляется переменная (`c`). Такого рода переменные, объявленные в теле функции, доступны только внутри тела функции, в которой были объявлены, и при завершении выполнения функции уничтожаются. Поэтому они носят название *локальных*.

Разумеется, в теле любой функции мы можем обратиться к любой переменной, объявленной вне ее тела (*глобальной*). Так, кстати, часто и делают.

Однако при этом нужно учитывать следующий важный момент. Если существуют две переменные с одинаковыми именами, одна — глобальная, другая — локальная, то при обращении по этому имени будет получен доступ *к локальной* переменной. Одноименная глобальная переменная будет «замаскирована» своей локальной «тезкой».

Вызов функций

После объявления функции ее можно *вызвать* из любого сценария, присутствующего на этой же странице. Формат вызова функции:

```
<имя функции>([<список передаваемых функции аргументов, разделенных
запятыми>])
```

Здесь указывается `имя` нужной функции и в круглых скобках через запятую приводятся значения передаваемых ей *`аргументов`*. *Аргументы* — это значения, которые будут присвоены параметрам этой функции и которые будут обрабатываться в коде ее тела. Если функция не принимает параметров, в выражении ее вызова обязательно должны быть указаны пустые круглые скобки.

Функция вернет результат, который можно присвоить переменной или использовать в выражении.

Вот пример вызова объявленной нами ранее функции `divide`:

```
d = divide(3, 2);
```

Здесь мы подставили в выражение вызова функции в качестве аргументов константы `3` и `2`.

А здесь мы выполняем вызов функции с переменными в качестве аргументов и используем возвращенный ей результат в выражении:

```
s = 4 * divide(x, r) + y;
```

Если функция принимает необязательные параметры, и нас удовлетворяют их значения по умолчанию, мы можем при вызове не указывать соответствующие им аргументы, все или некоторые из них. Например, функцию `divide` со вторым необязательным параметром мы можем вызвать так:

```
s = divide(4);
```

Тогда в переменной `s` окажется число `2` — результат деления `4` (значение первого аргумента) на `2` (значение второго, необязательного, параметра по умолчанию).

Если функция не возвращает результата, то ее вызывают так:

```
doSomething(1, 2, 3, 6);
```

Более того, так можно вызвать и функцию, возвращающую результат, который в этом случае будет отброшен. Такой способ вызова может быть полезен, если результат, возвращаемый функцией, нам не нужен.

А так вызывается функция, не принимающая параметров:

```
s = computeValue();
```

Присваивание функций. Функциональный тип данных

JavaScript позволяет выполнить один фокус — присвоить функцию переменной:

```
var someFunc;
someFunc = divide;
```

Здесь мы присвоили переменной `someFunc` объявленную ранее функцию `divide`. Заметим, что в этом случае справа от оператора присваивания указывается только имя функции без скобок и аргументов.

Впоследствии мы можем вызывать эту функцию, обратившись к переменной, которой она была присвоена:

```
c = someFunc(1, 5);
```

Здесь мы вызвали функцию `divide`, обратившись к переменной `someFunc`.

Функция сама по себе относится к *функциональному типу*. Это еще один тип данных, поддерживаемый JavaScript.

А теперь рассмотрим листинг 16.13. Здесь мы объявили функцию и сразу же присвоили ее переменной. Как видим, имени объявляемой функции мы не указали — в данном случае оно не нужно. Подобная функция — не имеющая имени — называется *анонимной* (а имеющая имя — *именованной*).

Листинг 16.13

```
var someFunc = function(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
}
```

Анонимную функцию мы также можем использовать в качестве аргумента при вызове другой функции. В дальнейшем, занимаясь практическим программированием, мы будем делать это очень часто.

Массивы

Массив — это пронумерованный набор переменных (*элементов*), фактически хранящийся в одной переменной. Доступ к отдельному элементу массива выполняется по его порядковому номеру, называемому *индексом*. А общее число элементов массива называется его *размером*.

> ***Внимание!***
>
> Нумерация элементов массива начинается с нуля.

Чтобы создать массив, достаточно присвоить любой переменной список его элементов, разделенных запятыми и заключенных в квадратные скобки:

```
var someArray = [1, 2, 3, 4];
```

Здесь мы создали массив, содержащий четыре элемента, и присвоили его переменной `someArray`. После этого мы можем получить доступ к любому из элементов по его индексу, указав его в квадратных скобках после имени переменной, в которой хранится массив:

```
a = massive[2];
```

В этом примере мы получили доступ к третьему элементу массива, чтобы присвоить его другой переменной. (Нумерация элементов массива начинается с нуля — помните об этом!)

Определять сразу все элементы массива необязательно:

```
var someArray2 = [1, 2, , 4];
```

Здесь мы пропустили третий элемент массива, в результате чего он будет содержать значение `undefined`.

При необходимости мы легко сможем добавить к массиву еще один элемент, просто присвоив ему требуемое значение:

```
someArray[4] = 9;
```

При этом будет создан новый, пятый по счету, элемент массива с индексом 4 и значением 9.

Можно даже сделать так:

```
someArray[7] = 9;
```

В этом случае будут созданы четыре новых элемента, и восьмой элемент получит значение 9. Пятый, шестой и седьмой элементы также будут созданы и получат значение `undefined`.

Мы можем присвоить любому элементу массива другой массив (или, как говорят программисты, создать *вложенный* массив):

```
someArray[2] = ["n1", "n2", "n3"];
```

После этого можно получить доступ к любому элементу вложенного массива, указав последовательно оба индекса: сначала — индекс во «внешнем» массиве, потом — индекс во вложенном:

```
str = someArray[2][1];
```

Переменная `str` получит в качестве значения строку, содержащуюся во втором элементе вложенного массива, — `"n2"`.

JavaScript не будет возражать, если мы создадим массив, вообще не содержащий элементов (пустой массив). Для этого достаточно присвоить любой переменной «пустые» квадратные скобки:

```
var someArray = [];
```

Разумеется, впоследствии мы можем наполнить этот массив элементами:

```
someArray[0] = 1;
someArray[1] = 2;
someArray[2] = 3;
```

Переменная, хранящая ссылку на массив, содержит данные *объектного типа*. Это последний тип данных, поддерживаемый JavaScript, который мы рассмотрим.

Массивы идеально подходят в тех случаях, когда нужно хранить в одном месте упорядоченный набор данных. Ведь массив фактически представляет собой одну переменную.

Ссылки. Пустая ссылка *null*

Осталось рассмотреть еще один момент, связанный с организацией доступа к данным. Это так называемые *ссылки* — своего рода указатели на массивы, хранящиеся в соответствующих им переменных.

Когда мы создаем массив, JavaScript выделяет под него область памяти и помещает в нее значения элементов этого массива. Но в переменную, которой мы присвоили вновь созданный массив, помещается не сама эта область памяти, а ссылка на нее. Если теперь обратиться к какому-либо элементу этого массива, JavaScript извлечет из переменной ссылку, по ней найдет нужную область памяти, вычислит местонахождение нужного элемента и выдаст его значение.

Далее, если мы присвоим значение переменной массива другой переменной, будет выполнено присваивание именно ссылки. В результате получатся две переменные, ссылающиеся на один и тот же массив.

Рассмотрим такой пример:

```
var myArray = ["p", "strong", "em"];
var newArray = myArray;
```

Здесь создается массив `myArray` с тремя элементами, после чего он присваивается переменной `newArray` — при этом последняя получает в качестве значения ссылку на массив. Если потом мы присвоим новое значение первому элементу массива `myArray`:

```
myArray[0] = "h1";
```

и обратимся к нему через переменную `newArray`:

```
s = newArray[0];
```

то в переменной `s` окажется строка `"h1"` — новое значение первого элемента этого массива. Как видим, переменные `myArray` и `newArray` указывают на один и тот же массив.

То же самое справедливо для функций и экземпляров объектов (о них разговор пойдет далее). То есть при присваивании функции переменной в последнюю заносится лишь ссылка на функцию.

> ***Внимание!***
>
> В дальнейшем для простоты мы будем считать, что в переменной хранится не ссылка на какую-либо сущность (функцию, массив или экземпляр объекта), а сама эта сущность. Если нужно будет специально указать, что переменная хранит ссылку, мы так и укажем.

JavaScript поддерживает особое значение `null`, также относящееся к объектному типу. Оно носит название *пустой ссылки* и означает, что хранящая ее переменная не указывает ни на какой массив, экземпляр объекта или функцию. Пустая ссылка применяется в практическом программировании довольно часто.

Объекты и экземпляры объектов

Итак, мы познакомились с типами данных, переменными, константами, операторами, выражениями, функциями и массивами. Но это была, так сказать, присказка, а сказка будет впереди. Настала пора узнать о самых сложных типах данных JavaScript — объектах.

Понятия объекта и экземпляра объекта

Объект — это сложная сущность, способная хранить произвольное количество значений разных типов. Для этого объект определяет набор своего рода внутренних переменных, называемых *свойствами*, — такое свойство может хранить одно значение любого типа: строку, число, логическую величину или массив.

А еще объект может содержать набор внутренних функций, называемых *методами*. Методы могут обрабатывать данные, хранящиеся в свойствах или полученные «извне», менять значения свойств и возвращать результат, как обычные функции.

Каждый объект должен иметь уникальное имя, по которому к нему можно обратиться, чтобы создать его экземпляр. К именам объектов предъявляют те же требования, что и к именам переменных и функций.

Объект — это всего лишь тип данных. Сущности же, хранящие реальные данные и созданные на основе этого объекта, называются его *экземплярами*. Точно так же, как строка `"JavaScript"` — экземпляр строкового типа данных, хранящий реальную строку.

Как уже упоминалось ранее, экземпляры объектов имеют такую же природу, как и массивы. Сам экземпляр объекта находится где-то в памяти компьютера, а в переменной хранится ссылка на него. Он, как и массив, принадлежит к объектному типу данных.

Создание экземпляра объекта

Чтобы создать экземпляр какого-либо объекта, мы воспользуемся *оператором создания экземпляра* `new`:

```
new <имя объекта>([<список аргументов, разделенных запятыми>])
```

Оператор `new` возвращает созданный экземпляр объекта. Его можно присвоить какой-либо переменной, свойству или передать в качестве аргумента функции или методу.

`Список аргументов` может как присутствовать, так и отсутствовать. Обычно он содержит значения, которые присваиваются свойствам экземпляра объекта при его создании.

Вот два примера создания экземпляров объектов:

- `var dNow = new Date();`

  Объект `Date`, предоставляемый самим языком JavaScript, хранит значение даты и времени и позволяет выполнять над ним различные манипуляции. Здесь мы соз-

даем экземпляр этого объекта, и, поскольку мы не указали никаких аргументов, он будет хранить текущие дату и время;

- `var dDate = new Date(2016, 10, 21);`

  А здесь мы указали в качестве аргументов значения года, порядкового номера месяца и число. В результате новый экземпляр объекта `Date` будет хранить дату 21 ноября 2016 года. (Нумерация месяцев начинается с нуля — так, ноябрь будет иметь номер 10.)

Работа с экземпляром объекта

Получив тем или иным способом экземпляр объекта, мы можем с ним работать: вызывать его методы и получать доступ к его свойствам. Для этого достаточно указать имя переменной, где хранится экземпляр объекта, поставить после него точку и записать имя требуемого свойства или метода. Разумеется, для вызова метода после его имени нужно указать набор передаваемых ему аргументов или хотя бы поставить круглые скобки.

Рассмотрим несколько примеров:

- `d = dDate.getDate();`

  Здесь мы обратились к переменной `dDate`, хранящей созданный ранее экземпляр объекта `Date`, и вызвали не принимающий аргументов метод `getDate`. Он вернет в качестве результата значение числа.

- `d = dDate.getMonth().toString();`

  А здесь мы сначала получаем номер месяца того же значения даты, вызвав не принимающий аргументов метод `getMonth`. А потом преобразуем полученный номер в строку, для чего вызываем уже у него также не принимающий аргументов метод `toString`. JavaScript позволяет писать такие «цепочки» методов (и свойств), каждый последующий из которых вызывается у результата, возвращенного предыдущим.

- `var pi = Math.PI;`

  Обращаемся к свойству `PI` объекта `Math`, где хранится значение числа π. (Кстати, это свойство — совершенно особое, и мы обязательно поговорим о нем позже.)

Рассмотренная нами запись «с точкой» применяется в большинстве случаев. Но JavaScript также поддерживает доступ к свойствам и методам экземпляра объекта посредством записи «с квадратными скобками». В этом случае после переменной ставятся квадратные скобки, в которые помещается строка с именем свойства или метода.

Так, в следующем примере мы извлекаем значение из свойства `prop` экземпляра объекта `oObj`:

```
n = oObj["prop"];
```

Добавленные свойства и методы

JavaScript позволяет нам добавить в экземпляр любого объекта новое свойство или метод, и это свойство (метод) будет принадлежать исключительно этому экземпляру объекта. Такие свойства и методы носят название *добавленных*.

Новое свойство добавляется присвоением ему значения. Если в качестве значения мы присвоим функцию, то получим метод. Такой трюк в некоторых случаях может заметно упростить программирование.

Вот пример создания добавленного свойства:

```
var d1 = new Date();
var d2 = new Date(2016, 10, 21);
d1.comment = "Текущая дата";
```

Здесь сначала создаются два экземпляра объекта `Date`, после чего у одного из них создается добавленное свойство `comment`, хранящее строку `"Текущая дата"`. Это свойство принадлежит только первому экземпляру объекта, но никак не второму.

Значение свойства `comment` у первого экземпляра объекта `Date` мы можем извлечь и, например, сохранить в другой переменной:

```
var c1 = d1.comment;
```

Но при попытке обратиться к этому свойству у второго экземпляра объекта:

```
var c2 = d2.comment;
```

мы получим ошибку выполнения сценария, поскольку такового свойства здесь нет.

Статические свойства и методы

И, наконец, существует ряд свойств и методов, которые принадлежат не экземпляру объекта, а самому объекту. Это *статические* свойства и методы.

Статические свойства и методы вызываются у самого объекта.

Вот пара примеров:

- `var a = Math.sqrt(2);`

  Объект `Math` встроен в JavaScript и поддерживает набор статических свойств, хранящих значения фундаментальных констант, и методов, выполняющих арифметические и алгебраические вычисления. В частности, его статический метод `sqrt` вычисляет квадратный корень от переданного ему аргумента.

- `var pi = Math.PI;`

  Уже знакомое нам свойство `PI` объекта `Math` также является статическим.

Встроенные объекты языка JavaScript

Что ж, с объектами и их экземплярами, свойствами и методами мы познакомились. Теперь давайте кратко «пробежимся» по объектам, которые поддерживаются самим языком JavaScript и которые мы будем использовать при написании сценариев.

Ранее мы познакомились со встроенным объектом `Date`, служащим для хранения значений даты и времени:

```
var dNow = new Date();
```

Объект `Date` поддерживает ряд методов, позволяющих получать отдельные составляющие даты и времени и манипулировать ими. Так, метод `getDate` возвращает число, `getMonth` — номер месяца, а `getFullYear` — год. Все эти методы не принимают аргументов, а возвращаемые ими результаты представляют собой числа.

В следующем примере мы объединяем в одну строку число, номер месяца, увеличенный на единицу, и год, разделяя их точками. Таким образом, мы получим значение даты в формате *<число>.<месяц>.<год>*. (JavaScript сам выполнит в этом случае неявное преобразование числовых величин в строки.):

```
var sNow = dNow.getDate() + "." + (dNow.getMonth() + 1) + "." +
dNow.getFullYear();
```

Объект `String` представляет строки. Да-да, обычная строка — это на самом деле экземпляр особого объекта!

```
var s = "JavaScript";
```

Мы только что создали экземпляр объекта `String`, хранящий строку `"JavaScript"`.

Свойство `length` объекта `String` хранит длину строки в символах:

```
var l = s.length;
var l = "JavaScript".length;
```

Оба этих выражения помещают в переменную `l` длину строки `"JavaScript"`.

Метод `substr` возвращает фрагмент строки заданной длины, начинающийся с указанного символа:

```
substr(<номер первого символа>[, <длина фрагмента>]);
```

Первым аргументом передается номер первого символа, включаемого в возвращаемый фрагмент строки. Второй, необязательный, аргумент задает длину возвращаемого фрагмента в символах — если он опущен, возвращаемый фрагмент будет содержать все оставшиеся символы строки.

> ***Внимание!***
>
> В JavaScript символы в строках нумеруются, начиная с нуля.

После выполнения этого сценария:

```
var s1 = s.substr(4);
var s2 = s.substr(4, 2);
```

в переменной `s1` окажется строка `"Script"`, а в переменной `s2` — строка `"Sc"`.

Объект `Number` представляет числа, а объект `Boolean` — логические величины:

```
var n = 123;
var b = false;
```

Числа и логические величины с точки зрения JavaScript также представляют собой экземпляры соответствующих объектов.

Все перечисленные ранее объекты поддерживают не принимающий аргументов метод `toString`, преобразующий значение, хранимое экземпляром объекта, в строку и возвращающий ее в качестве результата:

```
var pi = 3.14;
var s = pi.toString();
```

Выполнив это выражение, мы получим в переменной `s` строку `"3.14"`.

Объект `Array` служит для представления массивов:

```
var a = [1, 2, 3, 4];
```

Он поддерживает единственное свойство `length`, возвращающее размер массива, т. е. число элементов в нем:

```
var l = a.length;
var l = [1, 2, 3, 4].length;
```

Знакомый нам объект `Math`, как мы уже знаем, поддерживает набор статических методов для выполнения математических и тригонометрических вычислений. А объект `Function` представляет функции.

Мы не будем рассматривать здесь все встроенные объекты JavaScript и поддерживаемые ими свойства и методы (а их довольно много). Эти сведения можно найти в любой книге или на любом сайте, посвященном JavaScript-программированию.

Объект *Object* и использование его экземпляров

Но об одном встроенном объекте следует поговорить особо. Это объект `Object`, весьма специфический.

Экземпляры этого объекта обычно используются для хранения сложных структур данных. Для этого создается экземпляр объекта `Object`, и у него формируется набор добавленных свойств, в которых будут храниться необходимые значения, и методов, что будут ими манипулировать.

Экземпляры объекта `Object` удобно создавать с помощью особых выражений, называемых *инициализаторами*. Инициализатор чем-то похож на определение стиля (см. *главу 7*):

```
{
  <имя свойства 1>: <значение свойства 1>,
  <имя свойства 2>: <значение свойства 2>,
  . . .
  <имя свойства n-1>: <значение свойства n-1>;
  <имя свойства n>: <значение свойства n>
  <имя метода 1>: <функция, реализующая метод 1>,
  <имя метода 2>: <функция, реализующая метод 2>,
  . . .
```

```
  <имя метода n-1>: <функция, реализующая метод n-1>,
  <имя метода n>: <функция, реализующая метод n>
}
```

После выполнения инициализатора JavaScript вернет нам готовый экземпляр объекта `Object`, который мы можем присвоить какой-либо переменной или использовать в качестве аргумента функции или метода.

Листинг 16.14 представляет код, создающий экземпляр объекта `Object` со свойствами `a` и `b` и методом `m` и сохраняющий его в переменной `oObj`.

Листинг 16.14

```
var oObj = {
  a: 1,
  b: "JavaScript",
  m: function(x, y) { . . . }
};
```

Есть и другой способ создания экземпляров объекта `Object`. Сначала создается пустой массив. После этого у него формируются свойства в той же манере, как элементы у массива, только в качестве индексов указываются имена этих свойств. Листинг 16.15 показывает код, создающий экземпляр объекта, аналогичный рассмотренному ранее, но с применением такого способа.

Листинг 16.15

```
var oObj = [];
oObj["a"] = 1;
oObj["b"] = "JavaScript";
oObj["m"] = function(x, y) { . . . };
```

Забегая вперед, скажем, что именно в экземпляры объекта `Object` преобразуются данные, записанные в формате JSON. Об этом формате и вообще технологии AJAX разговор пойдет в *главе 22*.

Оператор *instanceof*

JavaScript поддерживает *оператор сравнения экземпляра объекта* `instanceof`. Он возвращает `true`, если экземпляр, указанный левым операндом, создан на основе объекта, указанного правым операндом, и `false` в противном случае.

Вот примеры:

- `var f1 = "JavaScript" instanceof String;`

  В переменной `f1` окажется значение `true`, поскольку строка `"JavaScript"` является экземпляром объекта `String`.

- `var f2 = 123 instanceof String;`

  А в переменной `f2` окажется значение `false`, поскольку число `123` не является экземпляром объекта `String`.

Цикл по свойствам объекта

Ранее мы рассмотрели три разновидности циклов, поддерживаемых JavaScript. Но существует еще одна разновидность — цикл по свойствам объекта, о котором нам следует знать.

Цикл по свойствам объекта выполняет свое тело для каждого свойства экземпляра объекта или элемента массива (как мы помним, массив в JavaScript тоже относится к объектному типу), что мы указали:

```
for (<переменная> in <массив>|<экземпляр объекта>) {
  <тело цикла>
}
```

Имя свойства (в случае экземпляра объекта) или индекс элемента (в случае массива) помещается в *переменную*, которая становится доступной в *теле цикла*. Там мы можем извлечь это имя или индекс и по нему получить значение свойства или элемента, чтобы использовать его в вычислениях.

После завершения перебора всех свойств или элементов выполнение цикла заканчивается и начинает обрабатываться следующий за ним код.

Листинг 16.16 показывает код, после выполнения которого в переменной `s` окажется строка с перечнем значений всех элементов массива `aTags`, разделенных запятыми.

Листинг 16.16

```
var aTags = ["<p>", "<strong>", "<em>"];
var s = "";
for (i in aTags) {
  if (s != "") s += ", ";
  s += aTags[i];
}
```

А код из листинга 16.17 сохранит в переменной `s` перечень позиций вида <*имя свойства экземпляра объекта* `oObj`>*:* <*значение свойства*>, также разделенных запятыми.

Листинг 16.17

```
var oObj = {a: 1, b: 2, c: "3"};
var s = "";
```

```
for (sn in oObj) {
  if (s != "") s += ", ";
  s += sn + ": " + oObj[sn].toString();
}
```

Цикл по свойствам объекта позволит нам быстро выполнить какое-либо действие для всех элементов коллекции (о коллекциях будет рассказано в *главе 17*). А в Web-программировании работать с коллекциями экземпляров объектов приходится довольно часто.

Обработка исключений

При выполнении Web-сценариев иногда возникают ошибки. В таком случае по умолчанию Web-обозреватель просто прерывает выполнение кода сценария, никак не сообщая о возникшей ошибке. Однако мы можем изменить это поведение, заставив его выполнить какие-либо другие действия, специально предусмотренные на случай возникновения ошибки, и тем самым все же выполнить сценарий.

Любая ошибка представляется в JavaScript экземпляром особого объекта. Он хранит сведения об ошибке, автоматически генерируется (или, как часто говорят программисты, *выбрасывается*) при ее возникновении и носит название *исключения*. (На практике слова «ошибка» и «исключение» часто используются как синонимы.)

Сгенерированное исключение можно обработать («поймать»), выполнив какие-либо действия по устранению причин, вызвавших ошибку, выводу на экран соответствующего сообщения и проч. Для этого применяется выражение обработки исключения следующего формата:

```
try {
  <блок try>
}
[catch(<параметр, которому будет присвоено исключение>) {
  <блок catch>
}]
[finally {
  <блок finally>
}]
```

- *Блок try* — это фрагмент кода, в котором может возникнуть ошибка.
- *Блок catch* — фрагмент кода, который будет выполнен при возникновении ошибки в *блоке try*. В нем можно получить доступ к исключению, представляющему ошибку, через параметр, имя которого указано в круглых скобках после ключевого слова `catch`. Если в *блоке try* не возникнет ошибок, *блок catch* никогда не будет выполнен.

  В виде *блока catch* оформляется код, который устраняет причины, вызвавшие ошибку, или, если это сделать невозможно, предпринимает некие альтернативные действия — например, выводит на экран соответствующее сообщение.

- *Блок finally* — фрагмент кода, который будет выполнен независимо от того, возникла в *блоке try* ошибка или нет. Если ошибка не возникла, он выполняется сразу после *блока try*, если возникла — после *блока catch*.

  В *блоке finally* обычно помещается код, выполняющий какие-либо завершающие задачи.

Блоки catch и *finally* являются необязательными. Однако на практике *блок catch* присутствует в выражении обработки исключения всегда (в отличие от *блока finally*).

Как уже говорилось, в *блоке catch* можно получить доступ к исключению, и извлечь из него дополнительные сведения о возникшей ошибке. Доступ к исключению производится через параметр, имя которого указывается после ключевого слова `catch` в круглых скобках.

JavaScript поддерживает следующие объекты-исключения:

- `Error` — базовый объект, от которого наследуют все более узкоспециализированные объекты, представляющие исключения конкретных типов (они приведены далее);
- `RangeError` — исключение, возникающее, если указанное значение не укладывается в допустимый диапазон;
- `ReferenceError` — исключение, возникающее при обращении к несуществующей переменной, необъявленной функции;
- `SyntaxError` — исключение, возникающее при ошибке в написании языковых конструкций JavaScript;
- `TypeError` — исключение, возникающее, если указанное значение не соответствует требуемому типу данных, а также при обращении к неподдерживаемому свойству или методу.

Все эти объекты поддерживают свойства `name` (имя объекта исключения) и `message` (текстовое описание возникшей ошибки).

Листинг 16.18 показывает пример обработки исключения. В блоке `try` выполняется обращение к несуществующему свойству `comment` экземпляра объекта `Date`, а в блоке `catch` из экземпляра объекта, представляющего исключение, извлекается имя этого объекта и текстовое описание ошибки.

Листинг 16.18

```
var oDate = new Date();
try {
  var s = oDate.comment;
} catch(exception) {
  var n = exception.name;
  var m = exception.message;
}
```

Комментарии JavaScript

Из *глав 2* и *7* мы знаем о существовании комментариев — особых фрагментов кода HTML и CSS, которые не обрабатываются Web-обозревателем и служат для того, чтобы Web-верстальщик смог оставить какие-либо заметки для себя или своих коллег. Было бы странно, если бы JavaScript не предоставлял аналогичной возможности.

Комментарий JavaScript, состоящий из одной строки, создают с помощью двух символов `//` (слеш), которые помещают в самом начале строки комментария:

```
// Это комментарий
var dNow = new Date();
```

Комментарий, состоящий из произвольного числа строк, создают с помощью последовательностей символов `/*` и `*/`. Между ними помещают строки, которые станут комментарием:

```
/*
  Это комментарий,
  состоящий из
  нескольких строк
*/
var dNow = new Date();
```

Как Web-сценарии помещаются в код Web-страницы?

Для вставки сценария в код страницы применяется парный тег `<script>`. Встретив его, Web-обозреватель поймет, что здесь присутствует сценарий, и его следует выполнить, а не выводить на экран.

Как говорилось в самом начале этой главы, сценарии, формирующие поведение страницы, можно поместить как в саму страницу, так и в отдельный файл. Рассмотрим, как это делается.

Если мы хотим поместить сценарий прямо в код страницы, создав *внутренний* сценарий, мы вставим его непосредственно в тег `<script>` (листинг 16.19).

Листинг 16.19

```
<script>
  var dNow = new Date();
  var sNow = dNow.getDate() + "." + dNow.getMonth() + "." +
  dNow.getFullYear();
  document.write(sNow);
</script>
```

Встретив такой тег, Web-обозреватель приостанавливает обработку HTML-кода страницы и начинает выполнять код сценария. А уже завершив его выполнение, продолжает обрабатывать код страницы.

Поскольку внутренние сценарии «принадлежат» самой странице, они обрабатываются очень быстро. Их недостаток проистекает из достоинства: если нам понадобится использовать один и тот же сценарий в нескольких страницах, мы должны вставить его в код каждой страницы, что долго и неудобно. Поэтому в качестве внутренних сейчас реализуют лишь совсем простые и экспериментальные сценарии.

Существует также возможность поместить сценарий в отдельный файл — *файл Web-сценария*, — создав *внешний* сценарий. Файлы сценариев представляют собой обычные текстовые файлы, содержат только код сценария без всяких тегов HTML и имеют расширение js.

Для вставки в страницу сценария, хранящегося в отдельном файле, применяется тег `<script>` такого вида:

```
<script src="<интернет-адрес файла сценария>" [defer] [async]></script>
```

Тег `<script>` оставляют пустым, и в него помещают обязательный в таком случае атрибут `src`, в качестве значения которого указывают интернет-адрес нужного файла сценария.

В листинге 16.20 представлен код файла сценария и тег, используемый для вставки его в страницу.

Листинг 16.20

```
// Файл сценария main.js
var dNow = new Date();
var sNow = dNow.getDate() + "." + (dNow.getMonth() + 1) + "." +
dNow.getFullYear();
document.write(sNow);

<!-- HTML-код страницы -->
<script src="main.js"></script>
```

Встретив такой тег `<script>`, Web-обозреватель также приостанавливает обработку HTML-кода страницы, загружает файл сценария и обрабатывает хранящийся в нем код.

Потребность в обработке кода сценария (который может быть весьма объемистым) может породить значительную задержку перед выводом страницы на экран. Однако мы можем изменить это поведение.

Если в тег `<script>` поместить атрибут без значения `defer`, Web-обозреватель выполнит загрузку и выполнение сценария только после завершения обработки кода

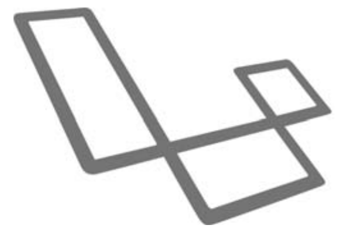
страницы и вывода ее на экран. В результате посетитель уже сможет начать просматривать страницу, пока код сценария будет обрабатываться.

А если указать в вышеупомянутом теге атрибут без значения `async`, обработка кода страницы и кода сценария будет вестись параллельно:

```
<script src="main.js" async></script>
```

Это позволит еще значительнее сократить, а то и полностью устранить задержку перед выводом страницы.

Внешние Web-сценарии можно применять сразу на нескольких страницах, задавая для них единообразное поведение. Поэтому они находят в настоящее время самое широкое применение.



ГЛАВА 17

Доступ к элементам страницы и управление ими

В предыдущей главе мы учили язык программирования JavaScript, на котором пишутся Web-сценарии, составляющие поведение страницы. Так что теперь мы можем приступить к Web-программированию.

Но каким образом мы можем управлять элементами страницы из Web-сценария? И вообще, как нам получить к ним доступ? Есть ли у нас для этого подходящие инструменты? В *главе 16* ни о чем таком не говорилось...

Объектная модель документа

Web-обозреватель, загрузив файл страницы и все сопутствующие файлы (изображения, аудио, видео, таблицы стилей, файлы сценариев) и прочитав ее HTML-код, формирует в памяти компьютера особое внутреннее представление этой страницы. Это представление является набором экземпляров специальных объектов, связанных друг с другом.

Сначала создается экземпляр объекта `HTMLDocument`, представляющий саму страницу и хранящий все ее параметры, — например, интернет-адрес, с которого она была загружена, и ее название (содержимое тега `<title>`).

После этого начинают создаваться экземпляры объектов, представляющие отдельные элементы страницы, — ее теги.

Первым создается экземпляр объекта `HTMLHtmlElement`, представляющий тег `<html>`. Что и не удивительно — ведь он находится на верхнем уровне иерархии тегов страницы, можно сказать, является ее «корнем».

Следующими формируются экземпляры объектов `HTMLHeadElement` и `HTMLBodyElement`, которые представляют, соответственно, секции заголовка и тела страницы — теги `<head>` и `<body>`. Между ними и созданным ранее экземпляром объекта `HTMLHtmlElement` устанавливается связь, сигнализирующая о том, что первые являются потомками второго.

За ними создаются экземпляры объектов, представляющие элементы следующего уровня вложенности. Так, для представления названия страницы (тега `<title>`) соз-

дается экземпляр объекта `HTMLTitleElement`, а для представления абзаца — экземпляр объекта `HTMLParagraphElement`. Между ними и их родителями также устанавливаются связи.

Для представления фрагментов текста, являющегося содержимым тегов, создаются экземпляры объекта `TextNode`. Разумеется, они также связываются с тегами — их родителями.

Все объекты, представляющие теги (`HTMLHtmlElement`, `HTMLBodyElement`, `HTMLParagraphElement` и т. д.), порождены от объекта более низкого уровня `HTMLElement`. Следовательно, они поддерживают все свойства и методы, предоставляемые этим объектом.

В качестве примера давайте рассмотрим нашу первую страницу, созданную еще в *главе 1* (ее код см. в листинге 1.1). Для нее Web-обозреватель сформирует структуру объектов, представленную в листинге 17.1 (отступы обозначают вложенность экземпляров объектов друг в друга, а в скобках приведены имена объектов, на основе которых созданы экземпляры).

Листинг 17.1

```
Сама страница (HTMLDocument)
  "Корень" страницы (HTMLHtmlElement)
    Секция заголовка (HTMLHeadElement)
      Метатег, задающий кодировку (HTMLMetaElement)
      Название страницы (HTMLTitleElement)
        Текст "Пример Web-страницы" (TextNode)
    Секция тела (HTMLBodyElement)
      Заголовок (HTMLHeadingElement)
        Текст "Изучаем язык HTML" (TextNode)
      Абзац (HTMLParagraphElement)
        Текст "Язык " (TextNode)
        Очень важный текст (HTMLStrongElement)
          Текст "HTML" (TextNode)
        Текст " предназначен для создания содержимого Web-страниц."
        (TextNode)
```

Обратим внимание, какая структура экземпляров объектов создается для содержимого абзаца. Сначала создается экземпляр объекта `TextNode`, представляющий текст, который предшествует тегу `<strong>`. Далее создается экземпляр объекта `HTMLStrongElement`, представляющий тег `<strong>`, вместе с потомком — экземпляром объекта `TextNode`, который представит содержимое этого тега. Наконец, формируется еще один экземпляр объекта `TextNode`, представляющий оставшееся содержимое абзаца.

Совокупность связанных друг с другом экземпляров объектов, представляющих страницу и ее элементы, носит название *объектной модели документа* (Document Object Model, *DOM*). Запомним эту аббревиатуру — она будет нам встречаться довольно часто.

Доступ к странице и ее элементам

Разобравшись с объектной моделью документа, приступим к изучению инструментов, предназначенных для доступа к самой странице и отдельным ее элементам.

Доступ к странице

Перед тем как начать выполнять код всех имеющихся на странице сценариев, JavaScript создает переменную `window`. В этой переменной хранится экземпляр объекта `Window`, представляющий текущее окно Web-обозревателя. А в свойстве `document` этого объекта хранится экземпляр объекта `HTMLDocument`, представляющий саму страницу.

В следующем примере мы сначала получаем доступ к странице, а потом извлекаем строку с ее названием (содержимым тега `<title>`):

```
var oDocument = window.document;
var sTitle = oDocument.title;
```

Весь код сценария выполняется «внутри» (или, другими словами, *в контексте*) экземпляра объекта `Window`, хранящегося в переменной `window`. Поэтому мы можем получать доступ к свойствам этого экземпляра объекта `Window` напрямую, не указывая переменную `window`. Это позволит несколько сократить объем кода:

```
var sTitle = document.title;
```

Так короче, не правда ли?

Доступ к элементам страницы

Получить доступ к отдельным элементам страницы тоже несложно. Для этого DOM предоставляет целый набор удобных инструментов.

Прямой доступ к элементу страницы

Прямой доступ к элементу страницы — по его имени, заданному атрибутом тега `id`, — можно получить посредством метода `getElementById` объекта `HTMLDocument`:

```
document.getElementById(<имя элемента>)
```

`Имя элемента` задается в виде строки. Метод возвращает экземпляр объекта, представляющего найденный элемент страницы, или `null`, если элемента с заданным именем на странице нет.

В листинге 17.2 приведен пример использования метода `getElementById` для доступа к элементу с указанным именем.

Листинг 17.2

```
<p>Первый абзац.</p>
<p id="p2">Второй абзац.</p>
```

```
<p>Третий абзац.</p>
. . .
var oP2 = document.getElementById("p2");
```

В переменной `oP2` окажется экземпляр объекта, представляющий абзац `p2` — второй по счету.

Доступ по имени тега или стилевого класса. Коллекции

Мы также можем получить доступ ко всем элементам, созданным с применением какого-либо тега. Для этого мы воспользуемся методом `getElementsByTagName`, поддерживаемым и объектом `HTMLDocument`, и объектом `HTMLElement`:

```
<страница или элемент>.getElementsByTagName(<имя тега>)
```

Имя тега указывается в виде строки без символов < и >.

А возвращаемое этим методом значение весьма примечательно. Это *коллекция* — экземпляр особого объекта, подобно массиву, хранящий список каких-либо значений. Для доступа к отдельным элементам такой коллекции используется привычный по массивам синтаксис (см. *главу 16*), а для получения размера коллекции — уже известное свойство `length`.

Итак, коллекция, возвращенная методом `getElementsByTagName`, содержит элементы, созданные с применением указанного тега. Если ни одного такого элемента на странице нет, будет возвращена пустая коллекция.

Пример из листинга 17.3 получает доступ ко всем абзацам страницы и привязывает к ним стилевой класс `paragraph`.

Листинг 17.3

```
var cPs = document.getElementsByTagName("p");
for (var i = 0; i < cPs.length; i++) {
  cPs[i].className = "paragraph";
}
```

Как уже говорилось, метод `getElementsByTagName` поддерживается не только страницей (объектом `HTMLDocument`), но и всеми объектами, порожденными от `HTMLElement`, — теми, что представляют ее элементы. Следовательно, мы можем вызвать этот метод у любого элемента страницы — в таком случае поиск будет производиться только среди его потомков.

Листинг 17.4 показывает, как можно это сделать. В результирующей коллекции окажутся только абзацы, вложенные в блок `b`.

Листинг 17.4

```
<p>Первый абзац.</p>
<div id="b">
  <p>Второй абзац.</p>
```

```
  <p>Третий абзац.</p>
</div>
. . .
var oB = document.getElementById("b");
var cPs = oB.getElementsByTagName("p");
```

Метод `getElementsByClassName`, также поддерживаемый объектами `HTMLDocument` и `HTMLElement`, позволяет найти элементы по имени привязанного к ним стилевого класса:

<страница или элемент>`.getElementsByClassName(`*<имя стилевого класса>*`)`

Имя стилевого класса указывается в виде строки без символа точки. В качестве результата также возвращается коллекция элементов.

Листинг 17.5 показывает пример использования этого метода — коллекция `cPars` будет содержать второй и третий абзацы.

Листинг 17.5

```
<p>Первый абзац.</p>
<p class="paragraph">Второй абзац.</p>
<p class="paragraph">Третий абзац.</p>
. . .
var cPars = document.getElementsByClassName("paragraph");
```

Доступ по селекторам CSS

Если нам нужно получить все элементы страницы, удовлетворяющие некому сложному условию, мы можем записать это условие в виде селектора CSS. О селекторах и правилах их написания говорилось в *главе 8*.

Для доступа к элементам страницы по заданному селектору применяются два метода, поддерживаемые объектами `HTMLDocument` и `HTMLElement`.

Метод `querySelector` возвращает первый из подходящих элементов или `null` в случае неудачного поиска:

<страница или элемент>`.querySelector(`*<селектор>*`)`

Селектор задается в виде строки.

Рассмотрим пару примеров:

- `var oEl = document.querySelector("nav a");`

  Получаем первую гиперссылку, находящуюся в семантической панели навигации.
- `var oEl = document.querySelector("#main h1 + p");`

  Получаем первый абзац, следующий за первым заголовком первого уровня, который находится в элементе с именем `main`.

Метод `querySelectorAll` возвращает коллекцию со всеми элементами, совпадающими с заданным селектором:

```
<страница или элемент>.querySelectorAll(<селектор>)
```

В следующем примере мы получаем все гиперссылки, находящиеся в семантической панели навигации:

```
var cEls = document.querySelectorAll("nav a");
```

Доступ к родителю, потомкам и соседним элементам. Узлы

Но предположим, что мы уже получили доступ к какому-либо элементу и теперь хотим добраться до его родителя и потомков. Как это сделать? Очень просто.

> ***Внимание!***
>
> Скажем сразу, что среди свойств, что мы рассмотрим далее, встречаются как доступные и для чтения, и для записи, так и те, что доступны лишь для чтения. Чтобы не возникало вопроса, какое свойство доступно для записи, мы будем в дальнейшем указывать, что доступное лишь для чтения свойство *возвращает* значение, а доступное и для чтения, и для записи — *хранит*.

Свойства `parentElement` и `parentNode` объекта `HTMLElement` (и, стало быть, всех объектов, представляющих элементы страницы) возвращают родителя текущего элемента.

После выполнения кода из листинга 17.6 в переменной `oParent` окажется блок, в который вложен абзац `p1`, — его родитель.

Листинг 17.6

```
<div>
  <p id="p1">Первый абзац.</p>
</div>
. . .
var oP1 = document.getElementById("p1");
var oParent = oP1.parentElement;
```

Для получения всех потомков элемента страницы мы можем использовать следующие два свойства, также поддерживаемые объектом `HTMLElement`:

- `children` — возвращает коллекцию всех тегов-потомков;
- `childNodes` — возвращает коллекцию всех потомков — и тегов, и фрагментов текста, и даже комментариев.

Коллекция, хранящаяся в свойстве `childNodes`, весьма специфическая. Ее элементы — и теги, и фрагменты текста, и комментарии — представлены в виде *узлов* — экземпляров объекта `Node`. Чтобы узнать, что представляет собой этот элемент коллекции, следует использовать особые свойства объекта `Node`, которые мы обязательно изучим позже.

И еще. Web-обозреватель считает узлами не только фрагменты текста, представляющие собой содержимое тегов, но и комбинации пробелов, табуляции, возвратов каретки и переводов строки, которыми при наборе HTML-кода обычно разделяются теги. Это может внести немалую путаницу.

В качестве примера сказанного рассмотрим код из листинга 17.7.

Листинг 17.7

```
<div id="b">
  <p>Первый абзац.</p>
  <p>Второй абзац.</p>
</div>
. . .
var oB = document.getElementById("b");
var cPs = oP1.childNodes;
```

По идее, после выполнения этого кода в переменной `cPs` должна оказаться коллекция из двух элементов — двух абзацев, вложенных в блок `b`. Однако эта коллекция будет включать в себя целых пять элементов (листинг 17.8).

Листинг 17.8

```
Блок (HTMLDivElement)
  Текст из символов возврата каретки, перевода строки и двух пробелов
  (TextNode)
  Абзац (HTMLParagraphElement)
    Текст "Первый абзац." (TextNode)
  Текст из символов возврата каретки, перевода строки и двух пробелов
  (TextNode)
  Абзац (HTMLParagraphElement)
    Текст "Второй абзац." (TextNode)
  Текст из символов возврата каретки и перевода строки (TextNode)
```

Поэтому, если нам требуется получить только теги-потомки, лучше обратиться к свойству `children`.

Объект `HTMLElement` поддерживает еще несколько свойств, позволяющих получить доступ к потомкам, — как тегам, так и узлам:

- `firstElementChild` — возвращает первый тег — потомок элемента или `null`, если элемент не имеет потомков;
- `firstChild` — возвращает первый узел — потомок элемента или `null`, если элемент не имеет потомков;
- `lastElementChild` — возвращает последний тег — потомок элемента или `null`, если элемент не имеет потомков;
- `lastChild` — возвращает последний узел — потомок элемента или `null`, если элемент не имеет потомков.

В следующем примере в переменной oP1 окажется первый абзац из вложенных в блок b:

```
var oP1 = oB.firstElementChild;
```

И, наконец, ряд свойств того же объекта позволит нам получить доступ к соседним элементам — тегам или узлам:

- previousElementSibling — возвращает предыдущий тег того же уровня вложенности или null, если таковой отсутствует;
- previousSibling — возвращает предыдущий узел того же уровня вложенности или null, если таковой отсутствует;
- nextElementSibling — возвращает следующий тег того же уровня вложенности или null, если таковой отсутствует;
- nextSibling — возвращает следующий узел того же уровня вложенности или null, если таковой отсутствует.

В следующем примере в переменной oP1 окажется предыдущий сосед блока b:

```
var oPS = oB.previousElementSibling;
```

Свойство ownerDocument объекта HTMLElement возвращает экземпляр объекта HTMLDocument, представляющий саму страницу (то же самое, что свойство document).

Быстрый доступ к элементам страницы

К ключевым элементам — тегам <html>, <head> и <body>, равно как и ко всем гиперссылкам, графическим изображениям, формам и якорям, — удобнее получать доступ через следующие свойства, поддерживаемые объектом HTMLDocument:

- documentElement — возвращает тег <html> (экземпляр объекта HTMLHtmlElement);
- head — возвращает тег <head> (экземпляр объекта HTMLHeadElement);
- body — возвращает тег <body> (экземпляр объекта HTMLBodyElement);
- anchors — возвращает коллекцию всех якорей (о якорях рассказывалось в *главе 5*);
- forms — возвращает коллекцию всех Web-форм;
- images — возвращает коллекцию всех изображений;
- links — возвращает коллекцию всех гиперссылок.

Вот пара примеров:

- `var oBody = document.body;`

  Получаем доступ к секции тела страницы.
- `var oForm1 = document.forms[0];`

  Получаем первую форму из всех присутствующих на странице.

Работа со страницей и ее элементами

Получив доступ к странице или ее элементу, мы можем считать их параметры или задать для них новые значения. Можно также указать новое содержимое для элемента страницы, и это содержимое может представлять собой как обычный текст, так и HTML-код. Мы даже имеем возможность создать на странице новые элементы и удалить уже имеющиеся.

Работа с параметрами страницы

Получить или задать параметры страницы мы можем, воспользовавшись свойствами объекта `HTMLDocument`:

- `title` — хранит название страницы в виде строки;
- `URL` — возвращает интернет-адрес страницы в виде строки;
- `referrer` — возвращает интернет-адрес страницы, с которой был выполнен переход на данную страницу, в виде строки;
- `domain` — возвращает доменное имя Web-сервера, с которого была загружена страница, в виде строки;
- `lastModified` — возвращает дату и время последнего изменения файла страницы в виде экземпляра объекта `Date` JavaScript (см. *главу 16*);
- `readyState` — возвращает строку с обозначением состояния загрузки страницы:
  - `"uninitialized"` — загрузка еще не началась;
  - `"loading"` — загрузка выполняется;
  - `"loaded"` — загрузка завершена, хотя, возможно, не полностью (внедренные элементы еще не загружены);
  - `"interactive"` — загрузка еще не завершена, но посетитель уже может просматривать страницу;
  - `"complete"` — загрузка полностью завершена.

В следующем примере мы добавляем к названию страницы ее интернет-адрес, взятый в скобки:

```
document.title = document.title + " (" + document.URL +")";
```

Работа с параметрами элемента

Разговор о работе с параметрами отдельного элемента страницы будет довольно долгим. Поэтому мы рассмотрим этот вопрос по частям.

Работа с основными параметрами

Для получения и задания основных параметров элемента страницы служат свойства объекта `HTMLElement`.

Рассмотрим их:

- `id` — хранит имя элемента (значение атрибута тега `id`);
- `className` — хранит привязанные к элементу стилевые классы (значение атрибута тега `class`);
- `title` — хранит текст всплывающей подсказки для элемента. (Ее также можно задать прямо в теге, в атрибуте `title`.)
- `tagName` — возвращает строку с именем тега, которым был создан элемент, приведенным к верхнему регистру;
- `nodeType` — возвращает тип узла в виде числа: 1 (тег), 3 (фрагмент текста) или 8 (комментарий HTML);
- `nodeName` — для узлов-тегов возвращает строку с именем тега, приведенным к верхнему регистру (как и свойство `tagName`), строку `"#text"` для узла — фрагмента текста и `"#comment"` для узла-комментария;
- `nodeValue` — возвращает содержимое узла — фрагмента текста и комментария и `null` для тега.

Листинг 17.9 показывает пример использования этих свойств.

Листинг 17.9

```
var oB = document.getElementById("b"), s;
if (oB.tagName.toLowerCase() == "div") {
  s = "Блок"
} else {
  s = "Не блок"
}
```

Здесь не принимающий аргументов метод `toLowerCase` объекта `String` приводит символы строки к нижнему регистру и возвращает преобразованную строку в качестве результата.

Работа с параметрами местоположения и размеров элемента страницы

Целый ряд свойств объекта `HTMLElement` позволяет узнать местоположение и размеры элемента страницы:

- `clientWidth` — возвращает ширину области блочного элемента, отведенную под вывод его содержимого, с учетом внутренних отступов слева и справа. Ширина вертикальной полосы прокрутки (если она есть), рамки и внешние отступы не учитываются. Значение этого свойства представляет собой число и измеряется в пикселах;
- `clientHeight` — возвращает высоту области блочного элемента, отведенную под вывод его содержимого, с учетом внутренних отступов сверху и снизу. Высота горизонтальной полосы прокрутки (если она есть), рамки и внешние отступы не

учитываются. Значение этого свойства представляет собой число и измеряется в пикселах;

- `offsetLeft` — возвращает горизонтальную координату левого верхнего угла элемента относительно элемента, хранящегося в свойстве `offsetParent`, в виде числа в пикселах;
- `offsetTop` — возвращает вертикальную координату левого верхнего угла элемента относительно элемента, хранящегося в свойстве `offsetParent`, в виде числа в пикселах;
- `offsetParent` — возвращает элемент, относительно которого рассчитываются значения свойств `offsetLeft` и `offsetTop`. Этот элемент представляет собой:
  - либо ячейку таблицы, если элемент, у которого вызывается это свойство, находится в ячейке таблицы;
  - либо ближайшего родителя элемента, у которого вызывается это свойство, являющегося позиционируемым (о позиционируемых элементах шла речь в *главе 13*);
  - либо секцию тела страницы;
  - если элемент, у которого вызывается это свойство, является фиксированным, всегда возвращается `null`;
- `offsetWidth` — возвращает ширину блочного элемента с учетом внутренних отступов слева и справа, ширины вертикальной полосы прокрутки (если она есть) и рамки, внешние отступы не учитываются. Значение этого свойства представляет собой число и измеряется в пикселах;
- `offsetHeight` — возвращает высоту блочного элемента с учетом внутренних отступов сверху и снизу, высоты горизонтальной полосы прокрутки (если она есть) и рамки, внешние отступы не учитываются. Значение этого свойства представляет собой число и измеряется в пикселах.

Следующие четыре свойства имеет смысл использовать только для элементов с прокруткой (они были описаны в *главе 12*):

- `scrollWidth` — возвращает полную ширину содержимого блочного элемента с учетом внутренних отступов слева и справа. Рамки и внешние отступы не учитываются. Значение этого свойства представляет собой число и измеряется в пикселах;
- `scrollHeight` — возвращает полную высоту содержимого блочного элемента с учетом внутренних отступов сверху и снизу. Рамки и внешние отступы не учитываются. Значение этого свойства представляет собой число и измеряется в пикселах;
- `scrollLeft` — хранит величину, на которую содержимое элемента было прокручено вправо, в виде числа в пикселах;
- `scrollTop` — хранит величину, на которую содержимое элемента было прокручено вниз, в виде числа в пикселах.

Пример использования некоторых из этих свойств приведен в листинге 17.10.

Листинг 17.10

```
section { height: 100px;
          overflow-y: auto;
          position: relative; }
. . .
<section>
  . . .
  <div id="somediv">. . .</div>
  . . .
</section>
. . .
var oSc = document.getElementsByTagName("section")[0];
var oDiv = document.getElementById("somediv");
oSc.scrollTop = oDiv.offsetTop + oDiv.offsetHeight;
```

Здесь мы прокручиваем содержимое тега `<section>` по вертикали таким образом, чтобы блок `somediv` появился в поле зрения посетителя. Отметим, что для того, чтобы вертикальная координата этого блока была вычислена относительно родителя (тега `<section>`), а не секции тела страницы, мы превратили родитель в относительно позиционируемый элемент.

Метод `getBoundingClientRect` объекта `HTMLElement` позволяет нам выяснить местоположение и размеры элемента относительно клиентской области окна Web-обозревателя. Он не принимает аргументов и возвращает экземпляр объекта `TextRectangle`, который поддерживает следующие свойства:

- `left` — возвращает горизонтальную координату верхнего левого угла;
- `top` — возвращает вертикальную координату верхнего левого угла;
- `right` — возвращает горизонтальную координату нижнего правого угла;
- `bottom` — возвращает вертикальную координату нижнего правого угла;
- `width` — возвращает ширину элемента;
- `height` — возвращает высоту элемента.

Все эти величины задаются в виде чисел и измеряются в пикселах.

В следующем примере мы получаем вертикальную координату блока `somediv` относительно клиентской области окна Web-обозревателя:

```
var tr = oDiv.getBoundingClientRect();
var t = tr.top;
```

Работа с атрибутами тега и их значениями

Для работы с атрибутами тега, с помощью которого создан элемент, и их значениями DOM предоставляет нам несколько методов объекта `HTMLElement`.

- Метод `getAttribute` позволяет нам получить значение заданного атрибута тега:

  `<элемент>.getAttribute(<имя атрибута тега>)`

  `Имя атрибута тега` задается как строка. Метод возвращает значение атрибута тега всегда в виде строки. Если указанный атрибут в теге не задан, возвращается пустая строка или `null`.

- Метод `setAttribute` задает для атрибута тега новое значение:

  `<элемент>.setAttribute(<имя атрибута тега>, <значение атрибута тега>)`

  `Значение атрибута тега` также задается в виде строки. Результата этот метод не возвращает.

- Метод `removeAttribute` удаляет из тега атрибут с заданным именем:

  `<элемент>.removeAttribute(<имя атрибута тега>)`

  Этот метод также не возвращает результата.

- Наконец, метод `hasAttribute` возвращает `true`, если тег содержит атрибут с заданным именем, и `false` в противном случае:

  `<элемент>.hasAttribute(<имя атрибута тега>)`

Пример использования этих методов можно увидеть в листинге 17.11. Представленный в нем код получает доступ к гиперссылке `home`, проверяет, указан ли в ее теге атрибут `target` и содержит ли он значение `"_blank"`, и, если так, удаляет этот атрибут тега.

Листинг 17.11

```
var oA = document.querySelector("a#home");
if ((oA.hasAttribute("target")) &&
  (oA.getAttribute("target") == "_blank"))
  oA.removeAttribute("target");
```

Работа со стилями

Разумеется, мы можем получить доступ из сценариев и к стилям, примененным к какому-либо элементу страницы.

Объект `HTMLElement` поддерживает свойство `style`. Оно возвращает экземпляр объекта `CSSStyleDeclaration`, представляющий встроенный стиль, который задан для этого элемента. (О встроенных стилях и их указании см. *главу 7*.)

Для доступа к отдельным атрибутам стиля объект `CSSStyleDeclaration` поддерживает большое количество соответствующих свойств. Имена этих свойств формируются по следующим правилам:

- все символы дефиса из имени атрибута стиля удаляются;
- буквы, следующие за символами дефиса, приводятся к верхнему регистру.

В табл. 17.1 приведены примеры формирования имен свойств объекта `CSSStyleDeclaration` на основе имен соответствующих им атрибутов стилей.

Таблица 17.1. *Примеры формирования имен свойств объекта* `CSSStyleDeclaration` *на основе имен соответствующих им атрибутов стилей*

Имя атрибута стиля	**Имя свойства**
position	position
background-color	backgroundColor
border-left-style	borderLeftStyle

Значения всех этих свойств задаются в том же формате, что и значения атрибутов стилей, которым они соответствуют.

В следующем примере мы задаем для секции тела страницы черный цвет фона:

```
document.body.style.backgroundColor = "black";
```

Код из листинга 17.12 превращает блок `somediv` в свободно позиционируемый и задает ему местоположение и размеры.

Листинг 17.12

```
var oDiv = document.getElementById("somediv");
oDiv.style.position = "absolute";
oDiv.style.left = "100px";
oDiv.style.top = "50px";
oDiv.style.width = "200px";
oDiv.style.height = "150px";
```

Но что делать, если нам нужно получить параметры результирующего стиля, созданного на основе всех стилей, что действуют на элемент? Воспользоваться весьма примечательным методом `getComputedStyle` объекта `Window`:

```
window.getComputedStyle(<элемент>, <псевдоэлемент>)
```

Элемент, для которого требуется получить результирующий стиль, задается в виде представляющего его экземпляра объекта. Если нужно получить результирующий стиль части содержимого этого элемента, вторым аргументом в виде строки следует указать соответствующий *псевдоэлемент*, — в противном случае в качестве второго аргумента нужно передать `null`. (О псевдоэлементах говорилось в *главе 8*.)

Метод `getComputedStyle` возвращает экземпляр объекта `CSSStyleDeclaration`, представляющий полученный результирующий стиль. Что нам и нужно.

Код из листинга 17.13 получает результирующий стиль, примененный к блоку `somediv`, извлекает из него горизонтальную координату этого блока, после чего смещает его вправо на 100 пикселов.

Листинг 17.13

```
var oDiv = document.getElementById("somediv");
var oStyle = window.getComputedStyle(oDiv, null);
var x = parseInt(oStyle.left) + 100;
oDiv.style.left = x + "px";
```

Функция `parseInt`, поддерживаемая JavaScript, преобразует переданную в качестве единственного параметра строку в число, которое и возвращает в качестве результата. Она справляется не только со строками, содержащими лишь число, но даже и со строками, которые начинаются с числа и продолжаются буквами. Дело в том, что значение атрибута стиля `left` (как и всех атрибутов стилей, задающих значения размеров и координат) заканчивается обозначением единицы измерения, и JavaScript преобразовать его в число автоматически не сможет.

Работа с содержимым элемента

Получить или изменить содержимое элемента страницы очень просто. Для этого предназначены два свойства объекта `HTMLElement`.

Свойство `innerHTML` хранит HTML-код, формирующий содержимое элемента. Мы можем использовать его для того, чтобы полностью заменить это содержимое новым.

В следующем примере мы вывели в абзац `output` текст, отформатированный с применением HTML-тегов, создав тем самым его новое содержимое:

```
<p id="output"></p>
. . .
var oOutput = document.getElementById("output");
oOutput.innerHTML = "<strong>Внимание!</strong> Выводим текст."
```

Свойство `textContent` хранит содержимое элемента в виде обычного текста:

- если попытаться получить значение свойства `textContent` у элемента, содержащего в качестве потомков, помимо текста, еще и теги, возвращенное значение будет представлять собой строку, составленную из текстового содержимого этого элемента и его тегов-потомков;
- если присвоить этому свойству в качестве нового значения какой-либо текст, все содержимое элемента, включая теги-потомки, будет заменено этим текстом. Если новое значение свойства содержит теги, последние будут просто выведены на экран.

Пример:

```
<p id="output"></p>
. . .
var oOutput = document.getElementById("output");
oOutput.textContent = "Внимание! Выводим текст."
```

Добавление нового содержимого

Если нам нужно без особых хлопот добавить на страницу новые элементы, мы можем воспользоваться методами `write` и `writeln` объекта `HTMLDocument`. Они выводят на страницу значения, переданные им в качестве аргументов, в том месте, где встретились. При этом метод `writeln` после выведенного значения вставляет символы возврата каретки и перевода строки:

```
document.write|writeln(<значение 1>, <значение 2>, . . . <значение n>)
```

Выводимых значений может быть сколько угодно, и все они будут выведены вплотную друг к другу.

Рассмотрим несколько примеров:

- строки "`Java`" и "`Script`" будут выведены друг за другом в одной строке, образуя слово "`JavaScript`":

  ```
  document.write("Java");
  document.write("Script");
  ```

- другой способ вывести слово "`JavaScript`", составленное из строк "`Java`" и "`Script`":

  ```
  document.write("Java", "Script");
  ```

- в этом случае слова "`Java`" и "`Script`" будут выведены через пробел. В пробел будут преобразованы Web-обозревателем символы возврата каретки и перевода строки, вставленные методом `writeln`:

  ```
  document.writeln("Java");
  document.writeln("Script");
  ```

Код из листинга 17.14 выводит на экран абзац, представленный в виде его HTML-кода.

Листинг 17.14

```
<p>Первый абзац.</p>
<script>
  document.write("<p>Второй абзац.</p>");
</script>
<p>Третий абзац.</p>
```

Добавление и удаление элементов страницы

Для добавления и удаления элементов страницы DOM предоставляет нам целый набор методов.

Первый метод — `createElement`, поддерживаемый объектом `HTMLDocument`. Он создает элемент страницы на основе указанного тега:

```
document.createElement(<имя тега>)
```

Имя тега указывается в виде строки без символов < и >. Метод возвращает экземпляр объекта, представляющий созданный элемент.

В следующем примере мы создаем графическое изображение и задаем его параметры, установив значения соответствующих атрибутов тега:

```
var oImg = document.createElement("img");
oImg.setAttribute("src", "/images/picture.svg");
oImg.setAttribute("alt", "Изображение");
```

Второй ключевой метод носит название `createTextNode` и также поддерживается объектом `HTMLDocument`. Он создает текстовый элемент на основе заданного нами текста:

```
document.createTextNode(<текст>)
```

Текст указывается в виде строки. Метод возвращает экземпляр объекта `TextNode`, представляющий созданный текстовый элемент.

Пример:

```
oText = document.createTextNode("Это абзац.");
```

Теперь нам нужно вывести созданный с помощью рассмотренных методов элемент на экран. Для этого нам следует вставить его в другой элемент страницы, сделав его потомком последнего. Эту задачу выполняют два метода объекта `HTMLElement`, которые мы сейчас рассмотрим.

Метод `appendChild` вставляет указанный элемент в самый конец элемента, у которого был вызван. В результате вставленный элемент станет последним потомком своего родителя:

```
<элемент — будущий родитель>.appendChild(<вставляемый элемент>)
```

Вставляемый элемент указывается в качестве экземпляра соответствующего объекта. Он же будет возвращен в качестве результата.

Код из листинга 17.15 создает блок, задает для него стилевой класс, помещает в него созданные ранее графическое изображение и, наконец, помещает готовый блок с изображением в самый конец страницы.

Листинг 17.15

```
var oDiv = document.createElement("div");
oDiv.className = "image";
oDiv.appendChild(oImg);
document.body.appendChild(oDiv);
```

Метод `insertBefore` вставляет новый элемент перед заданным нами элементом:

```
<элемент — будущий родитель>.insertBefore(<вставляемый элемент>[,
<элемент — потомок будущего родителя, перед которым будет вставлен новый
элемент>])
```

Все элементы также задаются в виде экземпляров объектов. Если второй аргумент не указан, новый элемент будет помещен в самый конец родителя — как и при вызове метода `appendChild`.

Осталось сказать, что метод `insertBefore` возвращает в качестве результата вставленный элемент.

Листинг 17.16 показывает код, создающий абзац, задающий для него имя `p1`, вставляющий в него созданный ранее текстовый элемент и помещающий на страницу перед абзацем `p2`.

Листинг 17.16

```
<body>
  <p id="p2">Это другой абзац.</p>
</body>
. . .
var oP1 = document.createElement("p");
oP1.id = "p1"
oP1.appendChild(oText);
var oP2 = document.getElementById("p2");
document.body.insertBefore(oP1, oP2);
```

Метод `replaceChild` объекта `HTMLElement` выполняет замену одного элемента на другой:

```
<элемент-родитель>.replaceChild(<заменяющий элемент>, <заменяемый
элемент>)
```

В качестве результата будет возвращен заменяемый элемент.

В следующем примере мы заменяем абзац `p2` созданным ранее блоком с изображением:

```
document.body.replaceChild(oDiv, oP2);
```

Если же нам понадобится удалить уже присутствующий на странице элемент, нам на помощь придет метод `removeChild` объекта `HTMLElement`:

```
<элемент-родитель>.removeChild(<удаляемый элемент>)
```

Удаленный элемент будет возвращен в качестве результата.

В следующем примере мы удаляем вставленный ранее на страницу абзац `p1`:

```
document.body.removeChild(oP1);
```

Еще нам может пригодиться метод `cloneNode` объекта `HTMLElement`. Он создает точную копию элемента, у которого был вызван:

```
<копируемый элемент>cloneNode([<копировать потомки?>])
```

Если в качестве аргумента указать значение `true`, то будет создана полная копия элемента вместе с его потомками. Если же передать методу значение `false` или во-

обще не указывать его аргумент, то будет скопирован лишь сам элемент без потомков. Возвращаемым результатом станет созданная копия элемента.

Код из листинга 17.17 делает копию первого присутствующего на странице абзаца, задает для него другое текстовое содержимое и добавляет в самый конец страницы.

Листинг 17.17

```
<body>
  <p>Это абзац.</p>
  . . .
</body>
. . .
var oP1 = document.body.querySelector("body p:first-of-type");
var oP2 = oP1.cloneNode();
oP2.textContent = "Это другой абзац";
document.body.appendChild(oP2);
```

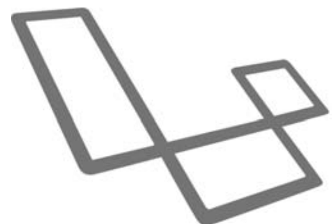
Последний полезный метод, который мы здесь рассмотрим, носит имя `isEqualNode`. Он поддерживается объектом `HTMLElement`, сравнивает элемент, у которого был вызван, с элементом, переданным единственным аргументом, и возвращает `true`, если эти элементы одинаковы, и `false` в противном случае:

```
<первый сравниваемый элемент>.isEqualNode(<второй сравниваемый элемент>)
```

Листинг 17.18 представляет код, выполняющий сравнение двух полностью одинаковых абзацев. В результате в переменной `bF` окажется значение `true`, поскольку оба сравниваемых абзаца абсолютно одинаковы.

Листинг 17.18

```
<p>Абзац.</p>
<p>Абзац.</p>
. . .
var cPs = document.getElementsByTagName("p");
var bF = cPs[0].isEqualNode(cPs[1]);
```



ГЛАВА 18

Обработка событий

Мы уже узнали, как получить доступ к странице и ее элементам, как получить и изменить их параметры, как заменить содержимое элемента и даже как добавить на страницу новые элементы и удалить уже существующие. И тем самым сделали очередной шаг в мир Web-программирования.

Однако все Web-сценарии, что мы писали в предыдущей главе, выполнялись непосредственно при загрузке страницы. В то время как бóльшая часть сценариев, что присутствуют на страницах, выполняются в ответ на какие-либо действия посетителя, — например, на щелчок на каком-либо элементе или при наведении на него курсора мыши. Как это сделать?

Введение в события и их обработку

Для таких случаев Web-обозреватель предоставляет нам исключительно мощный механизм событий.

События

Событием в терминологии программирования называется сообщение об изменении состояния какого-либо экземпляра объекта, представляющего страницу, ее элемент или сам Web-обозреватель (об объектах, представляющих Web-обозреватель, мы поговорим в *главе 20*). Цель события — дать возможность сценарию отреагировать на него.

Вот несколько примеров. Когда завершается загрузка страницы, она из состояния «еще не загружено» переходит в состояние «уже загружено», и генерируется событие, говорящее об окончании ее загрузки. Когда загруженный видеоролик начинает воспроизводиться, его состояние меняется от «ролик не воспроизводится» на «ролик воспроизводится», и возникает событие «воспроизведение ролика». Когда посетитель щелкает мышью на элементе страницы, его состояние меняется от «элемент, на котором не щелкнули мышью» на «элемент, на котором щелкнули мышью», и сигналом этого также станет соответствующее событие.

Каждое событие возникает в ответ на изменение строго определенного состояния элемента. Так, в ответ на щелчок мышью генерируется событие типа «щелчок мышью», а в ответ на запуск воспроизведения ролика — событие «воспроизведение ролика».

Каждое событие характеризуется набором определенных параметров. В эти параметры входят элемент страницы, в котором возникло событие (*источник события*), координаты точки, на которой посетитель щелкнул мышью, номер нажатой кнопки мыши и пр. Мы можем воспользоваться этими данными в сценарии.

Обработчики событий и их привязка

Ранее было сказано, что событие имеет целью дать сценарию средство отреагировать на его возникновение. Что это за средство? *Обработчик события.* Это обычная функция JavaScript, которая вызывается при наступлении определенного события в определенном элементе страницы. Обработчик, собственно, и реализует реакцию на событие.

Чтобы функция стала обработчиком события, нам следует выполнить ее *привязку*. Обработчик привязывается одновременно к событию, на которое следует реагировать (обрабатывать), и к элементу страницы, в котором может возникнуть это событие.

Привязку обработчика события выполняет метод `addEventListener`, поддерживаемый объектом `HTMLElement`:

```
<элемент страницы>.addEventListener(<событие>, <обработчик>[,
<обрабатывать на фазе туннелирования?>])
```

`Событие` задается в виде его наименования, указанного в строковом виде. В качестве *`обработчика`* можно указать как имя обычной именованной функции (без круглых скобок!), так и анонимную функцию.

Если третьим аргументом задано значение `true`, будет выполняться обработка события на фазе туннелирования. Если же передать третьим аргументом значение `false` или вообще опустить третий аргумент, обработка события будет выполняться на фазе всплытия. О фазах событий мы поговорим в конце этой главы, а пока что просто не будем указывать этот аргумент.

Метод `addEventListener` не возвращает результата.

Вот пара примеров:

- привязываем обработчик события `click`, возникающего при щелчке мышью, к элементу `active`:

  ```
  function activeClick() { . . . }
  . . .
  var oActive = document.getElementById("active");
  oActive.addEventListener("click", activeClick);
  ```

- а здесь мы используем в качестве обработчика события `click` анонимную функцию:

```
var oActive = document.getElementById("active");
oActive.addEventListener("click", function() { . . . });
```

Количество обработчиков, которые мы можем привязать к событиям одного элемента, не ограничено. Мы даже можем привязать несколько различных обработчиков к одному и тому же событию — при этом выполняться они будут в том же порядке, в котором были привязаны.

Если нам понадобится удалить привязку функции-обработчика к событию, мы вызовем метод `removeEventListener` объекта `HTMLElement`:

```
<элемент страницы>.removeEventListener(<событие>, <обработчик>[,
<обрабатывать на фазе туннелирования?>])
```

Отметим, что его вызов должен выполняться с теми же значениями аргументов, что и «парный» ему вызов метода `addEventListener`.

В следующем примере мы «отвязываем» от элемента `active` привязанную ранее к событию `click` функцию-обработчик `activeClick`:

```
oActive.removeEventListener("click", activeClick);
```

Существует еще один способ привязки обработчика к элементу страницы и событию. Он устарел и не совсем соответствует современным принципам программирования, однако поддерживается до сих пор и в некоторых случаях незаменим.

Для привязки обработчика к событию по такому способу в теге элемента, в котором требуется обрабатывать событие, создается следующий атрибут:

```
on<имя события>="<код обработчика>"
```

То есть код обработчика записывается непосредственно в теге нужного элемента.

В следующем примере мы указываем в качестве обработчика события `click` блока вызов функции `divClick`:

```
<div onclick="divClick();"> . . . </div>
```

Бо́льшая часть сценариев, которые мы напишем в процессе создания страниц, как раз и будут обработчиками различных событий.

Получение сведений о событии

Если нам нужно в теле функции-обработчика события получать какие-либо сведения о событии (например, элемент-источник), мы укажем в объявлении этой функции единственный параметр. Ему будет присвоен экземпляр особого объекта, хранящий сведения о событии.

Пример:

```
function activeClick(evt) { . . . }
. . .
var oActive = document.getElementById("active");
oActive.addEventListener("click", activeClick);
```

Здесь в объявлении функции-обработчика указан параметр `evt`, которому и будет присвоен экземпляр объекта, хранящий сведения о событии.

Сведения о событиях различных типов представляются разными объектами. Но все они порождены от объекта `Event`, хранящего базовые сведения.

Объект `Event` поддерживает следующие свойства:

- `target` — возвращает элемент-источник события (то есть элемент, в котором возникло это событие);
- `currentTarget` — возвращает элемент страницы, в котором выполняется обработка этого события, то есть элемент, к которому привязан этот обработчик события.

  Нужно отметить, что значения свойств `currentTarget` и `target` могут не совпадать. Более подробно об этом мы поговорим в конце этой главы, когда начнем изучать туннелирование и всплытие событий;
- `type` — возвращает имя события в виде строки (например, `"click"`);
- `timeStamp` — возвращает время возникновения события в виде количества миллисекунд, прошедших с 1 января 1970 года.

Объект `Event` также поддерживает еще несколько свойств и два метода, о которых мы поговорим в конце этой главы.

В листинге 18.1 показан код обработчика события `click`, указывающего для элемента страницы, на котором был выполнен щелчок мышью, красный цвет текста. Чтобы получить элемент, в котором возникло событие, использовалось свойство `target` объекта `Event`.

Листинг 18.1

```
function elClick(evt) {
  var el = evt.target;
  el.style.color = "red";
}
```

События, поддерживаемые элементами страницы

А теперь самое время познакомиться с событиями, поддерживаемыми элементами страницы. Существует несколько типов таких событий.

События мыши

События мыши возникают при манипуляциях с мышью: нажатии ее кнопок, перемещении ее и прокрутке колесика. Эти события описаны далее, все они всплывают и туннелируются, а их обработчик по умолчанию может быть отменен (о всплытии,

туннелировании событий и их обработчиках по умолчанию мы поговорим в конце этой главы).

- `click` — возникает при щелчке левой кнопкой мыши на элементе страницы;
- `dblclick` — возникает при двойном щелчке левой кнопкой мыши на элементе страницы;
- `contextmenu` — возникает при щелчке на элементе страницы правой кнопкой мыши перед выводом контекстного меню;
- `mousedown` — возникает сразу после нажатия любой кнопки мыши;
- `mouseup` — возникает при отпускании нажатой ранее кнопки мыши;
- `mousemove` — периодически возникает при перемещении курсора мыши над элементом страницы;
- `mouseover` — возникает при наведении курсора мыши на элемент страницы;
- `mouseenter` — то же самое, что `mouseover`, но не всплывает и не туннелируется;
- `mouseout` — возникает при уводе курсора мыши с элемента страницы;
- `mouseleave` — то же самое, что `mouseout`, но не всплывает и не туннелируется;
- `wheel` — возникает при вращении колесика мыши, когда ее курсор находится над элементом страницы.

Все события мыши, за исключением `wheel`, представляются объектом `MouseEvent`. Этот объект (как и все объекты-события) наследуется от объекта `Event` и, следовательно, поддерживает все его свойства.

Свойства объекта `MouseEvent` представлены в табл. 18.1.

Таблица 18.1. *Свойства объекта* `MouseEvent`

Свойство	Описание
`clientX`	Возвращает горизонтальную координату курсора мыши, вычисленную относительно левого верхнего угла клиентской области окна Web-обозревателя, в пикселах
`clientY`	Возвращает вертикальную координату курсора мыши, вычисленную относительно левого верхнего угла клиентской области окна Web-обозревателя, в пикселах
`pageX`	Возвращает горизонтальную координату курсора мыши, вычисленную относительно левого верхнего угла страницы, в пикселах
`pageY`	Возвращает вертикальную координату курсора мыши, вычисленную относительно левого верхнего угла страницы, в пикселах
`screenX`	Возвращает горизонтальную координату курсора мыши, вычисленную относительно левого верхнего угла экрана, в пикселах
`screenY`	Возвращает вертикальную координату курсора мыши, вычисленную относительно левого верхнего угла экрана, в пикселах
`relatedTarget`	Для событий `mouseover` и `mouseenter` возвращает элемент, с которого был уведен курсор мыши, для событий `mouseout` и `mouseleave` — элемент, на который был наведен курсор мыши

***Таблица 18.1** (окончание)*

Свойство	Описание
altKey	Возвращает true, если во время возникновения события была нажата клавиша <Alt>
ctrlKey	Возвращает true, если во время возникновения события была нажата клавиша <Ctrl>
shiftKey	Возвращает true, если во время возникновения события была нажата клавиша <Shift>
metaKey	Возвращает true, если во время возникновения события была нажата клавиша <Meta>[1]
button	Возвращает номер нажатой кнопки мыши: 0 (левая), 1 (средняя или колесико) или 2 (правая)
which	Возвращает номер нажатой кнопки мыши: 0 (никакая), 1 (левая), 2 (средняя или колесико) или 3 (правая)
detail	Возвращает количество щелчков мышью, выполненных в течение короткого промежутка времени. Для события dblclick всегда возвращает 2, для событий mouseover и mouseout — 0

Событие `wheel` представляется объектом `WheelEvent`. Из всех поддерживаемых им свойств для нас интересны два следующих:

- `deltaY` — возвращает величину, на которую была выполнена прокрутка по вертикали, в пикселах. При прокрутке вниз возвращается положительное значение, при прокрутке вверх — отрицательное;
- `deltaX` — возвращает величину, на которую была выполнена прокрутка по горизонтали, в пикселах. При прокрутке вправо возвращается положительное значение, при прокрутке влево — отрицательное. Это свойство актуально только для мышей, имеющих колесико для горизонтальной прокрутки.

В листинге 18.2 показан код, выводящий на экран координаты курсора мыши относительно страницы и сведения о том, была ли нажата клавиша <Shift>.

Листинг 18.2

```
<p id="coords"></p>
<p id="shift"></p>
. . .
var elCoords = document.getElementById("coords");
var elShift = document.getElementById("shift");
```

[1] <Meta> — специальная клавиша на некоторых клавиатурах. Обозначается сплошным ромбом ◆. Функционально аналогична клавише <Command> клавиатур Apple и имеет такое же расположение. При ее отсутствии она эмулируется клавишами <Alt> или <Windows> либо нажатием с последующим отпусканием клавиши <Esc>.

```
document.body.addEventListener("mousemove", function(evt) {
  elCoords.textContent = "[" + evt.pageX + ", " + evt.pageY + "]";
  elShift.textContent = (evt.shiftKey) ? "<Shift>" : "";
});
```

События клавиатуры

События клавиатуры возникают при нажатии клавиш на клавиатуре. Поддерживаются три события такого рода, все они всплывают и туннелируются, а их обработчик по умолчанию может быть отменен.

- `keypress` — возникает при вводе символа с клавиатуры, то есть при нажатии только алфавитно-цифровых клавиш. Если клавиша удерживается нажатой, возникает периодически;
- `keydown` — возникает при нажатии на клавиатуре любой клавиши;
- `keyup` — возникает при отпускании ранее нажатой клавиши.

Нужно отметить, что события клавиатуры поддерживаются только теми элементами страницы, которые могут принимать фокус ввода, а именно гиперссылками и элементами управления.

Все события клавиатуры представляются объектом `KeyboardEvent`. Поддерживаемые им свойства приведены в табл. 18.2.

Таблица 18.2. *Свойства объекта* `KeyboardEvent`

Свойство	Описание
`charCode`	В случае события `keypress` возвращает код в кодировке Unicode введенного символа, для событий `keydown` и `keyup` всегда возвращает 0
`keyCode`	В случае события `keypress` возвращает то же, что и свойство `charCode`, в случае событий `keydown` и `keyup` возвращает код в кодировке Unicode нажатой клавиши
`which`	То же, что и `keyCode`
`altKey`	Возвращает `true`, если во время возникновения события была нажата клавиша <Alt>
`ctrlKey`	Возвращает `true`, если во время возникновения события была нажата клавиша <Ctrl>
`shiftKey`	Возвращает `true`, если во время возникновения была нажата клавиша <Shift>

> ***Внимание!***
> Unicode-коды символов можно найти по интернет-адресу **http://www.w3schools.com/charsets/ref_html_utf8.asp**.

Листинг 18.3 представляет сценарий, который выводит на экран Unicode-коды занесенных в поле ввода символов, разделяя их пробелами.

Листинг 18.3

```
<form><input id="txtChars"></form>
<p id="chars"></p>
. . .
var txtChars = document.getElementById("txtChars");
var elChars = document.getElementById("chars");
txtChars.addEventListener("keypress", function(evt) {
  elChars.textContent += evt.charCode + " ";
});
```

Событие прокрутки

Событие `scroll` возникает при прокрутке содержимого элемента. Понятно, что его имеет смысл обрабатывать только у элементов с прокруткой. Это событие не всплывает, не туннелируется, а его обработчик по умолчанию не может быть отменен.

Событие прокрутки представляется объектом `Event`.

События секции тела страницы

Секция тела (тег `<body>`) — весьма специфический элемент страницы. Наряду с событиями мыши, описанными ранее, она поддерживает набор событий, которые приведены далее. Они не всплывают, не туннелируются, а их обработчики по умолчанию не могут быть отменены.

- `load` — возникает сразу после окончания загрузки страницы;
- `hashchange` — возникает при переходе на другой якорь;
- `beforeunload` — возникает непосредственно перед выгрузкой страницы.

  Если из обработчика этого события вернуть в качестве результата строку, на экране появится окно-предупреждение с этой строкой и кнопками **ОК** и **Отмена**. Выгрузка страницы будет выполнена только в том случае, если посетитель нажмет кнопку **ОК**. Это можно использовать, чтобы спросить посетителя, действительно ли он желает покинуть текущую страницу;
- `unload` — возникает сразу после выгрузки страницы;
- `resize` — возникает при изменении размеров окна Web-обозревателя, в котором отображается страница.

Все эти события представляются объектом `Event`.

> ***Внимание!***
>
> Для привязки обработчиков указанных событий к секции тела следует использовать устаревший способ — посредством атрибутов тега вида `on`*`<имя события>`*. При использовании метода `addEventListener` привязка производится успешно, но обработчик почему-то не вызывается.

Вот пример использования события `beforeunload` для вывода окна-предупреждения с запросом на уход с текущей страницы:

```
<body onbeforeunload="return 'Вы действительно хотите покинуть эту ↳
страницу?';"
```

Особые случаи обработки событий

А теперь рассмотрим два особых случая обработки событий. Применяются они на практике нечасто, однако их знание позволит существенно упростить программирование.

Туннелирование и всплытие событий

Когда мы щелкаем на каком-либо элементе страницы мышью, в этом элементе возникает событие `click`. Казалось бы, все просто. Но не совсем.

Многие (но не все!) события при возникновении проходят три фазы: туннелирования, источника и всплытия, причем эти фазы проходят в том порядке, в котором они здесь указаны.

- Во время фазы *туннелирования* событие сначала возникает в самом внешнем элементе страницы (это тег `<html>`). Далее оно возникает в секции тега страницы в элементе (теге `<body>`), который является потомком самого внешнего тега, потом — в его потомке, который является родителем для элемента-источника, и т. д., пока не доберется до непосредственного родителя элемента-источника.

  Говоря другими словами, на фазе туннелирования событие проникает вглубь страницы по направлению от тегов меньшего уровня вложенности к тегам большего уровня вложенности, постепенно приближаясь по иерархии элементов к источнику, и, наконец, возникает в непосредственном родителе элемента-источника события.
- Во время фазы *источника* событие возникает в самом элементе-источнике.
- Во время фазы *всплытия* событие вновь возникает в элементе — непосредственном родителе элемента-источника, потом — в родителе родителя и т. д., пока не возникнет в самом внешнем элементе — теге `<html>`. То есть при всплытии событие проходит путь, обратный пути туннелирования.

Как уже было сказано, всплывают и туннелируются многие события, но не все. Так, событие `click` всплывает и туннелируется, а событие `resize` — нет.

Мы имеем возможность обработать событие во время фазы туннелирования или всплытия, привязав обработчик к какому-либо из родителей элемента-источника. В ряде случаев это может быть полезно.

Давайте вспомним формат вызова метода `addEventListener`:

```
<элемент страницы>.addEventListener(<событие>, <обработчик>[,
<обрабатывать на фазе туннелирования?>])
```

Фаза события, на которой должен быть выполнен обработчик, задается третьим, необязательным, аргументом этого метода, которым должна быть логическая величина:

- `false` (или если третий аргумент вообще не указан) — обработчик выполнится на фазе всплытия;
- `true` — обработчик выполнится на фазе туннелирования.

В следующем примере мы привязываем функцию `someFunc` к элементу `el` в качестве обработчика события `click` таким образом, чтобы событие обрабатывалось на фазе туннелирования:

```
el.addEventListener("click", someFunc, true);
```

Отметим, что, если обработчик привязывается непосредственно к элементу-источнику события (то есть событие обрабатывается на фазе источника, как мы делали это ранее), третий аргумент метода `addEventListener` не играет никакой роли. Указывать третий аргумент имеет смысл только в том случае, если обработчик привязывается к одному из родителей элемента-источника.

Обработку событий на фазе туннелирования применяют крайне редко и в очень специфических случаях. А вот обработка на фазе всплытия, напротив, используется очень часто. Типичный случай — обработка одного и того же события, возникающего в нескольких элементах страницы, в их родителе. Листинг 18.4 показывает код, выполняющий такую обработку.

Листинг 18.4

```
<div id="somediv">
  <p>Первый абзац</p>
  <p>Второй абзац</p>
  <p>Третий абзац</p>
</div>
. . .
var elSomeDiv = document.getElementById("somediv");
elSomeDiv.addEventListener("click", function(evt) {
  evt.target.style.color = "red";
});
```

Здесь при щелчке на любом из трех абзацев, вложенных в блок `somediv`, цвет его текста меняется на красный. Чтобы получить доступ к элементу-источнику события, применяется свойство `target` объекта `Event`, рассмотренное нами в начале этой главы.

Объект `Event`, представляющий событие, поддерживает два полезных свойства:

- `bubbles` — возвращает `true`, если это событие может всплывать и туннелироваться, и `false` в противном случае;

- `eventPhase` — возвращает числовое обозначение фазы, которую в данный момент проходит это событие:
  - 1 — фаза туннелирования;
  - 2 — фаза источника;
  - 3 — фаза всплытия.

И еще раз отметим, что свойство `target` возвращает элемент-источник события, а свойство `currentTarget` — элемент, в котором оно в данный момент обрабатывается (к которому привязан обработчик, в данный момент получающий доступ к этому свойству). Это могут быть разные элементы страницы.

Иногда возникает необходимость запретить дальнейшее прохождение события. Для этого достаточно вызвать в его обработчике не принимающий аргументов и не возвращающий результата метод `stopPropagation` объекта `Event`. Пример:

```
function someEventHandler(evt) {
  // Выполняем обработку события
  evt.stopPropagation();
}
```

Обработчик события по умолчанию и его отмена

За многими событиями закреплен *обработчик по умолчанию*, который предоставляется самим Web-обозревателем и выполняется после завершения работы всех остальных обработчиков. Так, при щелчке мышью на гиперссылке обработчик по умолчанию выполняет переход по указанному в теге гиперссылки интернет-адресу, при щелчке на кнопке — операцию, закрепленную за кнопкой (отправку данных или очистку формы).

Если нам не нужно, чтобы такой обработчик по умолчанию выполнялся, мы можем его отменить. Для этого достаточно вызвать не принимающий аргументов и не возвращающий результата метод `preventDefault` объекта `Event`. Пример:

```
function someEventHandler(evt) {
  // Выполняем обработку события
  evt.preventDefault();
}
```

Более приближенный к жизни пример — запрет вызова контекстного меню по нажатию правой кнопки мыши:

```
document.body.addEventListener("contextmenu", function(evt) {
  evt.preventDefault();
});
```

Не все события позволяют отменить обработчик по умолчанию. Так, обработчик по умолчанию для события `unload`, возникающего перед выгрузкой страницы, отменить нельзя — это сделано ради безопасности.

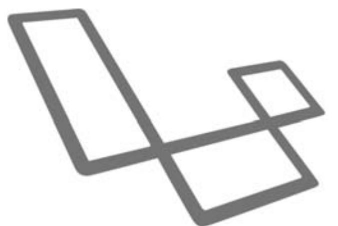
Свойство `cancellable` объекта `Event` возвращает `true`, если обработчик по умолчанию для этого события можно отменить, и `false` в противном случае.

В листинге 18.5 представлен код, отменяющий у события обработчик по умолчанию только в том случае, если это можно сделать.

Листинг 18.5

```
function someEventHandler(evt) {
  // Выполняем обработку события
  if (evt.cancellable)
    evt.preventDefault();
}
```

И наконец, свойство `defaultPrevented` объекта `Event` возвращает `true`, если обработчик этого события по умолчанию был отменен в одном из ранее выполнившихся обработчиков, и `false` в противном случае.



ГЛАВА 19

Управление интерактивными и внедренными элементами

Умение писать обработчики событий — ключевое в JavaScript-программировании. Овладев им, можно приступать к написанию вполне функциональных Web-сценариев, выполняющих какие-либо полезные действия.

Но для полноценной работы нам нужно узнать еще довольно много. Например, какие особенности по сравнению с обычными элементами страницы имеют элементы *интерактивные* (гиперссылки, формы и элементы управления) и *внедренные* (изображения, аудио- и видеоролики). Или, если конкретнее, какие дополнительные программные инструменты они нам предоставляют.

Интерактивные элементы

Начнем мы с интерактивных элементов: гиперссылок, Web-форм и элементов управления, поскольку элементами такого рода приходится управлять из сценариев очень часто.

Гиперссылки

Гиперссылки (точнее, представляющий их объект `HTMLAnchorElement`) поддерживают расширенный, по сравнению с рассмотренными в *главе 17*, набор свойств.

Свойства, характерные для гиперссылок, приведены в табл. 19.1. Большая их часть хранит фрагменты интернет-адреса гиперссылки (см. *главу 1*), и все они возвращают строковые значения, если иное не указано явно.

Таблица 19.1. *Свойства, характерные для гиперссылок*

Свойство	Описание
`href`	Хранит интернет-адрес целиком (соответствует атрибуту тега `href`)
`target`	Хранит цель гиперссылки (соответствует атрибуту тега `target`)
`protocol`	Хранит обозначение протокола. Если таковой не указан, для страниц, загруженных с Web-сервера, хранит строку `"http:"`, для страниц, загруженных с локального диска, — строку `"file:"`

Таблица 19.1 *(окончание)*

Свойство	Описание
host	Хранит комбинацию интернет-адреса Web-сервера (имени хоста) и номера порта (если он указан). Если таковые не указаны, хранит интернет-адрес, с которого была загружена страница. Если страница хранится на локальном диске, хранит пустую строку
hostname	Хранит интернет-адрес Web-сервера (имени хоста). Если таковой не указан, хранит интернет-адрес, с которого была загружена страница. Если страница хранится на локальном диске, хранит пустую строку
port	Хранит номер порта или пустую строку, если он не указан
pathname	Хранит путь к запрашиваемому файлу или символ слеша, если таковой не указан
search	Хранит набор GET-параметров или пустую строку, если он отсутствует
hash	Хранит имя якоря вместе с символом # или пустую строку, если таковое отсутствует
tabIndex	Хранит номер в порядке обхода в виде числа (соответствует атрибуту тега tabindex)
accessKey	Хранит «горячую» клавишу (соответствует атрибуту тега accesskey)

Гиперссылки поддерживают события мыши и клавиатуры, описанные в *главе 18*, и, помимо них, следующие события:

- focus — возникает при получении гиперссылкой фокуса ввода;
- blur — возникает при потере гиперссылкой фокуса ввода.

Эти события не всплывают, не туннелируются, и обработчики событий по умолчанию у них не могут быть отменены.

В следующем примере в переменной sHN окажется строка "www.w3c.org" — имя хоста сайта организации W3C:

```
<a id="wwwc" href="http://www.w3c.org/">W3C</a>
. . .
var oA = document.getElementById("wwwc");
var sHN = oA.hostname;
```

Web-формы

Форма представляется объектом HTMLFormElement. Этот объект поддерживает расширенный набор свойств, методов и событий.

Свойства, характерные для формы, приведены в списке далее и, в большинстве своем, предназначены для доступа к параметрам формы, заданным различными атрибутами тега <form>. Все эти свойства хранят строковые значения, если иное не указано явно.

- `elements` — возвращает коллекцию всех элементов управления, присутствующих в форме. Отметим, что элементы других типов (абзацы, блоки и т. п.) в эту коллекцию не попадут.
- `length` — возвращает количество элементов управления в форме в виде числа.
- `action` — хранит интернет-адрес серверной программы, которой будут отправлены введенные данные (соответствует атрибуту тега `action`).
- `target` — хранит цель, куда будут выведены результаты, возвращенные серверной программой (соответствует атрибуту тега `target`).
- `enctype` — хранит наименование метода кодирования данных (соответствует атрибуту тега `enctype`).
- `method` — хранит наименование метода отправки данных (соответствует атрибуту тега `method`).
- `name` — хранит имя формы, заданное атрибутом тега `name`.
- `autocomplete` — хранит признак, указывающий, работает ли в форме автозаполнение, в виде строки `"on"` или `"off"` (соответствует атрибуту тега `autocomplete`).
- `noValidate` — хранит признак того, будет ли форма выполнять проверку введенных в нее данных на корректность (соответствует атрибуту тега `novalidate`). Значение `true` активизирует проверку данных, значение `false` отключает ее.

Формы также поддерживают два дополнительных метода. Оба они не принимают аргументов и не возвращают результата:

- `submit` — инициирует отправку данных серверной программе;
- `reset` — выполняет очистку формы.

Отправка данных формой будет выполнена лишь в том случае, если во всех элементах управления, содержащихся в форме, указаны корректные данные. В противном случае отправка выполнена не будет, а рядом с элементами управления, куда были занесены некорректные данные, появятся всплывающие сообщения с описанием допущенных ошибок.

Формы поддерживают и два дополнительных события:

- `submit` — возникает непосредственно перед отправкой данных серверной программе;
- `reset` — возникает при выполнении очистки формы.

Эти события возникают только при нажатии кнопки отправки данных и кнопки очистки соответственно. При вызове описанных ранее методов `submit` и `reset` они не генерируются. Оба события всплывают, туннелируются, а обработчики по умолчанию для них могут быть отменены.

Элементы управления

Объекты, представляющие элементы управления, поддерживают довольно большое количество дополнительных свойств, методов и событий.

Все дополнительные свойства элементов управления представлены в списке далее. Они также, по большей части, позволяют получить доступ к значениям параметров элемента, заданных одноименными атрибутами тега.

- `value` — хранит текущее значение, занесенное в элемент управления, в виде строки.
- `defaultValue` — хранит изначальное значение элемента управления, заданное атрибутом тега `value`, в виде строки.
- `disabled` — хранит признак того, доступен ли элемент управления в настоящий момент: `true` — недоступен, `false` — доступен (соответствует атрибуту тега `disabled`).
- `readOnly` — хранит признак того, доступен ли элемент управления только для чтения: `true` — доступен только для чтения, `false` — доступен для чтения и ввода (соответствует атрибуту тега `readonly`).
- `required` — хранит признак того, является ли элемент управления обязательным для ввода: `true` — является, `false` — не является (соответствует атрибуту тега `required`).
- `name` — хранит имя элемента управления, заданное атрибутом тега `name`, в виде строки.
- `form` — возвращает форму, в которой находится элемент управления.
- `selectionStart` — хранит номер первого символа выделенного в поле ввода или области редактирования фрагмента текста в виде числа. Если текст не выделен, возвращает номер символа, на котором стоит текстовый курсор.
- `selectionEnd` — хранит номер последнего символа выделенного фрагмента текста в виде числа. Если текст не выделен, возвращает номер символа, на котором стоит текстовый курсор.
- `maxLength` — хранит максимальную длину значения, которое можно занести в поле ввода, в виде числа (соответствует атрибуту тега `maxlength`).
- `pattern` — хранит шаблон, с которым будет сверяться заносимое в поле ввода значение, в виде строки (соответствует атрибуту тега `pattern`).
- `placeholder` — хранит текст подсказки, выводимой прямо в поле ввода или области редактирования, в виде строки (соответствует атрибуту тега `placeholder`).
- `min` — хранит минимальное значение, которое можно указать в поле ввода числа или регуляторе, в виде числа (соответствует атрибуту тега `min`).
- `max` — хранит максимальное значение, которое можно указать в поле ввода числа или регуляторе, в виде числа (соответствует атрибуту тега `max`).
- `step` — хранит интервал между значениями, допустимыми для занесения в поле ввода числа или выбора с помощью регулятора, в виде числа (соответствует атрибуту тега `step`).
- `checked` — хранит текущее состояние флажка или переключателя: `true` — установлен, `false` — сброшен.

- `defaultChecked` — возвращает изначальное состояние флажка или переключателя, заданное атрибутом тега `checked`: `true` — установлен, `false` — сброшен.
- `selectedIndex` — хранит номер выбранного в списке пункта в виде числа. Если список позволяет выбирать одновременно несколько пунктов, возвращает номер первого выбранного пункта. Если ни один пункт не выбран, возвращает значение `-1`.
- `options` — возвращает коллекцию пунктов списка.
- `length` — возвращает количество пунктов в списке в виде числа.
- `multiple` — в случае списка хранит признак, указывающий, позволяет ли список выбрать сразу несколько пунктов, в случае поля ввода файла — признак, указывающий, позволяет ли поле выбрать сразу несколько файлов: `true` — позволяет, `false` — не позволяет (соответствует атрибуту тега `multiple`).
- `size` — возвращает размер списка в пунктах в виде числа (соответствует атрибуту тега `size`).
- `files` — возвращает коллекцию файлов, выбранных в поле ввода файла (о работе с этой коллекцией будет рассказано в *главе 21*).
- `accept` — хранит описание типов файлов, доступных для выбора в поле ввода файла (соответствует атрибуту тега `accept`).
- `autofocus` — хранит признак того, должен ли этот элемент управления получить фокус ввода сразу после загрузки страницы (соответствует атрибуту тега `autofocus`). Значение `true` активизирует автоматическое получение фокуса ввода, значение `false` деактивирует его.
- `accessKey` — хранит «горячую» клавишу для элемента управления в виде строки (соответствует атрибуту тега `accesskey`).
- `tabIndex` — хранит номер элемента управления в порядке обхода в виде числа (соответствует атрибуту тега `tabindex`).
- `autocomplete` — хранит признак, указывающий, разрешено ли для поля ввода автозаполнение, в виде строки `"on"` или `"off"` (соответствует атрибуту тега `autocomplete`).
- `type` — возвращает значение атрибута `type` тега `<input>`, т. е. тип элемента управления, в виде строки. Для области редактирования возвращает строку `"textarea"`, для списков с возможностью выбора одного пункта — `"select-one"`, для списков с возможностью выбора нескольких пунктов — `"select-multiple"`.
- `validationMessage` — возвращает текст сообщения о некорректно занесенном в элемент управления значении в виде строки.
- `validity` — возвращает сведения об ошибках, допущенных посетителем при занесении значения в элемент управления, в виде экземпляра объекта `ValidityState`. Этот объект поддерживает следующие доступные только для чтения свойства:

   - `patternMismatch` — `true`, если введенное значение не совпадает с заданным шаблоном;
   - `rangeOverflow` — `true`, если введенное число больше указанного максимально допустимого;
   - `rangeUnderflow` — `true`, если введенное число меньше указанного минимально допустимого;
   - `stepMismatch` — `true`, если введенное число не укладывается в заданный интервал;
   - `tooLong` — `true`, если введенная строка слишком длинная;
   - `typeMismatch` — `true`, если введенное значение не является числом, интернет-адресом или адресом электронной почты;
   - `valueMissing` — `true`, если не указано обязательное для ввода значение;
   - `customError` — `true`, если было задано собственное сообщение об ошибке ввода данных (как это сделать, будет описано чуть позже);
   - `valid` — `true`, если введенное значение полностью корректно (и все описанные ранее свойства хранят значение `false`).
- `willValidate` — возвращает `true`, если значение, присутствующее в этом элементе управления, будет проверяться на корректность. Для недоступных и доступных только для чтения элементов возвращается `false`.

В следующем примере мы получаем текст, занесенный в поле ввода:

```
<input id="txtName">
. . .
var txtName = document.getElementById("txtName");
var sName = textName.value;
```

В листинге 19.1 приведен код, делающий поле ввода доступным при установке флажка.

Листинг 19.1

```
<input type="checkbox" id="chkEnable" checked>
<input id="txtName">
. . .
var chkEnable = document.getElementById("chkEnable");
var txtName = document.getElementById("txtName");
chkEnable.addEventListener("click", function() {
  txtName.disabled = !chkEnable.checked;
});
```

Листинг 19.2 представляет пример кода, определяющего, были ли в поле ввода числа занесены корректные значения, и, если это не так, выводящего на экран соответствующее сообщение.

Листинг 19.2

```
<form>
  <input type="number" id="txtYear" min="2000" max="3000" step="10">
  <input type="button" id="btnCheck" value="Проверить">
</form>
<div id="output"></div>
. . .
var txtYear = document.getElementById("txtYear");
var btnCheck = document.getElementById("btnCheck");
var oOutput = document.getElementById("output");
btnCheck.addEventListener("click", function() {
  if (txtYear.validity.rangeUnderflow)
    oOutput.textContent = "Минимальное значение года - 2000";
  else if (txtYear.validity.rangeOverflow)
    oOutput.textContent = "Максимальное значение года - 3000";
  else if (txtYear.validity.stepMismatch)
    oOutput.textContent = "Значение года должно быть кратно 10";
  else
    oOutput.textContent = "";
});
```

Пункт списка поддерживает следующие дополнительные свойства:

- `selected` — хранит признак, выбран ли этот пункт в данный момент времени: `true` — выбран, `false` — не выбран;
- `text` — хранит текст пункта списка;
- `value` — хранит значение пункта списка, заданное атрибутом `value` тега `<option>`.
- `label` — хранит сокращенную версию текста пункта в виде строки (соответствует атрибуту тега `label`);
- `disabled` — хранит признак, доступен ли пункт списка для выбора: `true` — недоступен, `false` — доступен (соответствует атрибуту тега `disabled`);
- `defaultSelected` — возвращает изначальное состояние пункта списка, заданное атрибутом `selected` тега `<option>`: `true` — выбран, `false` — не выбран;
- `index` — хранит порядковый номер пункта в списке в виде числа;
- `form` — возвращает форму, в которой находится список, являющийся владельцем этого пункта.

После выполнения кода, приведенного в листинге 19.3, массив `a1` будет хранить текст всех выбранных в списке пунктов, а массив `a2` — их значения.

Листинг 19.3

```
<select id="selOptions" multiple>
  <option value="html">HTML</option>
```

```
  <option value="css">CSS</option>
  <option value="js">JavaScript</option>
  <option value="php">PHP</option>
  <option value="laravel">Laravel</option>
</select>
. . .
var selOptions = document.getElementById("selOptions");
var opts = selOptions.options;
var a1 = [];
var a2 = [];
for (var i = 0; i < opts.length; i++) {
  if (opts[i].selected) {
    a1.push(opts[i].text);
    a2.push(opts[i].value);
  }
}
```

Метод `push` объекта `Array` добавляет в массив новый элемент, переданный ему единственным аргументом.

Теперь перейдем к методам, поддерживаемым элементами управления. Сначала рассмотрим служебные методы, не принимающие аргументов (первые три метода не возвращают результата):

- `focus` — устанавливает фокус ввода на элемент управления;
- `blur` — удаляет фокус ввода с элемента управления и переносит его на следующий в порядке обхода элемент;
- `select` — выделяет все содержимое поля ввода или области редактирования;
- `checkValidity` — возвращает `true`, если в элемент управления формы занесено корректное значение, и `false` в противном случае.

Два специфических метода поддерживаются списком:

- метод `add` позволяет добавить в список новый пункт:

  ```
  add(<добавляемый пункт>[, <номер, который он займет в списке>])
  ```

  `Добавляемый пункт` указывается в виде экземпляра объекта, созданного методом `createElement` (см. *главу 17*). Если *`номер`* не указан, пункт будет добавлен в конец списка. Результата этот метод не возвращает;
- метод `remove` удаляет из списка пункт с заданным *`номером`*:

  ```
  remove(<номер удаляемого пункта>)
  ```

  Он также не возвращает результата.

Имеем в виду, что нумерация пунктов списка начинается с нуля.

Листинг 19.4 представляет код, сначала добавляющий в начало списка `lstOptions` новый пункт `"JSON"`, а потом удаляющий из него предпоследний пункт.

Листинг 19.4

```
var opt = document.createElement("option");
opt.textContent = "JSON";
lstOptions.add(opt, 0);
lstOptions.remove(lstOptions.length - 2);
```

Еще один метод, поддерживаемый элементами управления, мы рассмотрим чуть позже.

Элементы управления поддерживают события мыши и клавиатуры (см. *главу 18*). Помимо этого, ими поддерживаются следующие события:

- `change` — возникает, если значение, занесенное в элемент управления (для полей ввода, области редактирования и регулятора), или его состояние (для флажка и переключателя) изменено. У поля ввода, области редактирования и регулятора возникает сразу после того, как этот элемент потеряет фокус ввода, у флажков и переключателей — непосредственно в момент изменения состояния;
- `input` — возникает, когда значение, занесенное в элемент управления, изменяется. В отличие от события `change`, возникает сразу же после изменения значения, а не после того, как элемент управления потеряет фокус ввода, и только в полях ввода, областях редактирования и регуляторах.

  Эти два события всплывают, туннелируются, но их обработчики по умолчанию не могут быть отменены;
- `focus` — возникает при получении элементом управления фокуса ввода;
- `blur` — возникает при потере элементом управления фокуса ввода;
- `select` — возникает при выделении текста в поле ввода или области редактирования.

  Эти три события не всплывают, не туннелируются, и их обработчики по умолчанию не могут быть отменены.

Событие `input` можно использовать, чтобы реализовать проверку данных на корректность прямо в процессе их ввода. Если данные некорректны, следует указать для элемента управления, в который они занесены, собственное сообщение об ошибке — тогда при попытке отправить некорректные данных оно будет выведено на экран. Если же данные корректны, нужно указать в качестве сообщения об ошибке пустую строку.

Собственное сообщение об ошибке указывается с помощью метода `setCustomValidity`:

```
<элемент управления>.setCustomValidity(<сообщение об ошибке ввода>)
```

`Сообщение об ошибке ввода` указывается в виде строки. Результата этот метод не возвращает.

Листинг 19.5 показывает пример использования события `input` и метода `setCustomValidity`. Выполняется проверка на корректность значения года, занесен-

ного в поле ввода, и, если оно меньше 2000, для этого элемента управления указывается соответствующее сообщение об ошибке.

Листинг 19.5

```
<input type="number" id="txtYear">
. . .
var txtYear = document.getElementById("txtYear");
txtYear.addEventListener("input", function() {
  var n = parseInt(txtYear.value), s = "";
  if (!isNaN(n)) {
    if (n < 2000) {
      s = "Минимальное значение года - 2000";
    }
  }
  txtYear.setCustomValidity(s);
});
```

Функция `isNaN`, поддерживаемая JavaScript, возвращает `true`, если ей в качестве единственного аргумента передано значение `NaN` (не число). Эта проверка необходима, поскольку функция `parseInt` возвращает `NaN`, если не может преобразовать переданную ей строку в число (это может случиться, если посетитель занес в поле ввода что-либо, отличное от числа).

Внедренные элементы

Внедренные элементы — изображения, аудио- и видеоролики — также поддерживают расширенный по сравнению с прочими элементами набор свойств, методов и событий.

Графические изображения

Дополнительные свойства, поддерживаемые графическими изображениями, приведены в списке далее. Среди них есть весьма полезные — в частности, позволяющие узнать размеры изображения, хранящегося в указанном файле.

- `src` — хранит интернет-адрес файла изображения в виде строки (соответствует атрибуту тега `src`).
- `alt` — хранит текст замены в виде строки (соответствует атрибуту тега `alt`).
- `naturalWidth` — возвращает ширину изображения, загруженного из указанного файла, в пикселах в виде числа.
- `naturalHeight` — возвращает высоту изображения, загруженного из указанного файла, в пикселах в виде числа.
- `complete` — возвращает `true`, если изображение полностью загружено, и `false` в противном случае.

Рассмотрим пару примеров:

- выводим в элементе графического изображения другой файл:

```
<img id="activeImage" src="/images/picture1.jpg">
. . .
var oAI = document.getElementById("activeImage");
oAI.src = "/images/picture2.jpg";
```

- если ширина изображения, загруженного из графического файла, превышает ширину внутренней области родителя, отведенной под вывод его содержимого, устанавливаем ширину изображения равной ширине внутренней области родителя:

```
<img id="image" src="/images/picture.jpg">
. . .
var oI = document.getElementById("image");
if (oI.parentElement.clientWidth < oI.naturalWidth)
  oI.style.width = oI.parentElement.clientWidth + "px";
```

Еще графическое изображение поддерживает два дополнительных события, которые могут нам пригодиться:

- `load` — возникает сразу по окончании загрузки графического файла;
- `error` — возникает при прерывании загрузки графического файла в результате ошибки сценария или сетевого сбоя.

Аудио- и видеоролики

Элементы аудио- и видеороликов также поддерживают расширенный набор программных инструментов. И набор этот весьма велик.

Начнем, как обычно, с перечня дополнительных свойств:

- `src` — хранит интернет-адрес воспроизводящегося в данный момент ролика в виде строки;
- `volume` — хранит громкость звука. Значение должно представлять собой число с плавающей точкой от 0 (звук отсутствует) до 1 (максимальная громкость);
- `muted` — хранит признак, указывающий, приглушен ли звук в данный момент: `true` — приглушен, `false` — не приглушен (соответствует атрибуту тега `muted`);
- `currentTime` — хранит текущую позицию воспроизведения ролика в виде числа в секундах;
- `videoWidth` — возвращает ширину видеоролика, извлеченную из файла, в виде числа в пикселах;
- `videoHeight` — возвращает высоту видеоролика, извлеченную из файла, в виде числа в пикселах;
- `duration` — возвращает продолжительность ролика в виде числа в секундах;

- `paused` — хранит признак, указывающий, приостановлено ли воспроизведение ролика в данный момент: `true` — приостановлено, `false` — не приостановлено;
- `ended` — возвращает `true`, если воспроизведение ролика уже закончилось, и `false` в противном случае;
- `seeking` — возвращает `true`, если посетитель в данный момент меняет позицию воспроизведения ролика, и `false` в противном случае;
- `playbackRate` — хранит текущую скорость воспроизведения ролика. Значение должно представлять собой число с плавающей точкой, которое будет умножено на значение скорости воспроизведения, полученное из файла с роликом, положительные значения задают воспроизведение в прямом порядке, отрицательные — в обратном;
- `autoplay` — хранит признак, указывающий, запустится ли воспроизведение ролика сразу после его загрузки: `true` — запустится, `false` — не запустится (соответствует атрибуту тега `autoplay`);
- `controls` — хранит признак, указывающий, будут ли на экране присутствовать элементы управления воспроизведением ролика: `true` — будут присутствовать, `false` — не будут присутствовать (соответствует атрибуту тега `controls`);
- `loop` — хранит признак, указывающий, будет ли ролик воспроизводиться бесконечно: `true` — ролик воспроизводится бесконечно, `false` — ролик будет воспроизведен всего один раз (соответствует атрибуту тега `loop`);
- `poster` — хранит интернет-адрес файла заставки в виде строки (соответствует атрибуту тега `poster`);
- `preload` — хранит признак, указывающий, выполнять ли предварительную загрузку ролика, в виде строки (соответствует атрибуту тега `preload`);
- `currentSrc` — возвращает интернет-адрес воспроизводящегося в данный момент ролика в виде строки. Может отличаться от интернет-адреса, заданного в самом теге, если ролик предоставляется серверной программой, автоматически выбирающей мультимедийный файл в зависимости от параметров клиентского компьютера;
- `networkState` — возвращает признак того, выполняется ли загрузка ролика, в виде числа:
  - 0 — загрузка не выполняется, ролик не инициализирован;
  - 1 — загрузка не выполняется, но ролик уже инициализирован (вероятно, загружен полностью);
  - 2 — загрузка выполняется;
  - 3 — файл с роликом не найден;
- `readyState` — возвращает состояние готовности ролика к воспроизведению в виде числа:
  - 0 — либо ролик не указан, либо по какой-то причине не удается получить состояние его готовности;

- 1 — загружен лишь заголовок файла, хранящий сведения о самом ролике;
- 2 — загружены данные, достаточные для воспроизведения текущего кадра, но для воспроизведения следующего кадра данных недостаточно;
- 3 — загружены данные, достаточные для воспроизведения текущего и, по меньшей мере, следующего кадров;
- 4 — загружены данные, достаточные для бесперебойного воспроизведения ролика.

Листинг 19.6 показывает код, зацикливающий ролик и задающий для него половинную громкость.

Листинг 19.6

```
<video id="vid" src="film.mp4"></video>
. . .
var oVid = document.getElementById("vid");
oVid.loop = true;
oVid.volume = 0.5;
```

Ролики поддерживают три полезных метода, не принимающие аргументов и не возвращающие результата:

- `play` — запускает или возобновляет воспроизведение ролика;
- `pause` — приостанавливает воспроизведение ролика;
- `load` — выполняет перезагрузку ролика.

Что касается специфических для аудио- и видеороликов событий, то их список весьма велик:

- `loadstart` — возникает сразу после начала загрузки ролика;
- `durationchange` — возникает после получения в составе метаданных ролика его продолжительности;
- `loadedmetadata` — возникает сразу после загрузки всех метаданных ролика;
- `loadeddata` — возникает сразу после загрузки первого кадра ролика;
- `progress` — периодически возникает, когда Web-обозреватель загружает следующие за первым кадры ролика;
- `canplay` — возникает, когда Web-обозреватель решает, что он может, по крайней мере, начать воспроизведение ролика, однако в дальнейшем воспроизведение может быть приостановлено для подгрузки данных;
- `canplaythrough` — возникает, когда Web-обозреватель решает, что он может начать воспроизведение ролика без приостановок для подгрузки данных.

  Все указанные события возникают в начале загрузки ролика и именно в том порядке, в котором они были приведены;
- `play` — возникает при запуске или возобновлении воспроизведения ролика;

- `timeupdate` — периодически возникает в процессе воспроизведения ролика;
- `pause` — возникает при приостановке воспроизведения ролика (постановке его на паузу);
- `ended` — возникает по окончании воспроизведения ролика;
- `seeking` — возникает перед началом изменения текущей позиции воспроизведения ролика;
- `seeked` — возникает по окончании изменения текущей позиции воспроизведения ролика;
- `volumechange` — возникает при изменении громкости, а также при отключении и включении звука;
- `ratechange` — возникает при изменении скорости воспроизведения ролика;
- `waiting` — возникает при приостановке воспроизведения ролика на подгрузку последующих кадров;
- `playing` — возникает, как только ролик снова начинает воспроизводиться после приостановки на подгрузку данных;
- `suspend` — возникает, когда загрузка ролика приостанавливается, а также по ее завершению;
- `stalled` — возникает при прерывании загрузки ролика, спустя три секунды после получения последней порции данных (например, при исчезновении сетевого соединения);
- `error` — возникает при прерывании загрузки ролика вследствие ошибки.

Все эти события не всплывают, не туннелируются, и их обработчики по умолчанию не могут быть отменены. Они представляются объектом `Event` и, соответственно, не предоставляют каких-либо дополнительных параметров. Исключением является событие `progress`, представляемое объектом `ProgressEvent`, который поддерживает три следующих свойства:

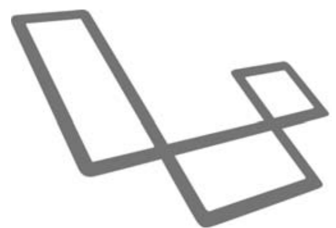
- `lengthComputable` — возвращает `true`, если Web-обозреватель может определить размер загружаемого мультимедийного файла, и `false` в противном случае;
- `loaded` — возвращает размер уже загруженной части файла в виде числа в байтах;
- `total` — возвращает полный размер загружаемого файла в виде числа в байтах.

Листинг 19.7 представляет реализацию мультимедийного проигрывателя, который использует для управления воспроизведением видеоролика собственные элементы управления: кнопки и регулятор.

Листинг 19.7

```
<video id="video" src="video.mp4"></video>
<form>
  <div><input id="sldTime" type="range" min="0" value="0" disabled></div>
```

```
  <div><input id="btnPlay" type="button" value="Пуск" disabled>
  <input id="btnPause" type="button" value="Пауза" disabled>
  <input id="btnStop" type="button" value="Стоп" disabled></div>
</form>
. . .
var video = document.getElementById("video");
var sldTime = document.getElementById("sldTime");
var btnPlay = document.getElementById("btnPlay");
var btnPause = document.getElementById("btnPause");
var btnStop = document.getElementById("btnStop");
video.addEventListener("loadedmetadata", function() {
  sldTime.max = video.duration;
  sldTime.disabled = false;
  btnPlay.disabled = false;
});
video.addEventListener("play", function() {
  btnPlay.disabled = true;
  btnPause.disabled = false;
  btnStop.disabled = false;
});
video.addEventListener("pause", function() {
  btnPlay.disabled = false;
  btnPause.disabled = true;
});
video.addEventListener("ended", function() {
  sldTime.value = 0;
  btnPlay.disabled = false;
  btnPause.disabled = true;
});
video.addEventListener("timeupdate", function() {
  sldTime.value = video.currentTime;
});
btnPlay.addEventListener("click", function() {
  video.play();
});
btnPause.addEventListener("click", function() {
  video.pause();
});
btnStop.addEventListener("click", function() {
  video.pause();
  video.currentTime = 0;
  btnStop.disabled = true;
});
```



ГЛАВА 20

Работа с Web-обозревателем

В предыдущих главах мы управляли исключительно страницей и ее элементами: задавали их параметры, создавали и удаляли элементы и реагировали на происходящие в них события. А что же сам Web-обозреватель? Можем ли мы управлять им?

Разумеется, можем. Для этого Web-обозреватель предоставляет набор созданных им самим экземпляров объектов, составляющих *объектную модель Web-обозревателя* (Browser Object Model, *BOM*), с которой мы и будем разбираться в этой главе.

Окна Web-обозревателя

Окно Web-обозревателя представляется объектом `Window`. Экземпляр этого объекта, являющийся текущим окном, хранится в созданной самим Web-обозревателем переменной `window`.

Поскольку весь код сценария выполняется в контексте экземпляра объекта `Window`, представляющего текущее окно (об этом уже говорилось в *главе 17*), мы можем при работе со свойствами и методами окна опускать переменную `window`. Это позволит несколько сократить объем кода.

Начнем, как обычно, с рассмотрения свойств, поддерживаемых окнами (табл. 20.1).

Таблица 20.1. *Свойства окна Web-обозревателя*

Свойство	Описание
`scrollX`	Возвращает величину, на которую открытая в окне страница была прокручена по горизонтали, в виде числа в пикселах
`scrollY`	Возвращает величину, на которую открытая в окне страница была прокручена по вертикали, в виде числа в пикселах
`pageXOffset`	То же, что и `scrollX`
`pageYOffset`	То же, что и `scrollY`

Таблица 20.1 *(окончание)*

Свойство	Описание
`innerWidth`	Возвращает ширину клиентской области окна в виде числа в пикселах
`innerHeight`	Возвращает высоту клиентской области окна в виде числа в пикселах
`outerWidth`	Возвращает полную ширину окна в виде числа в пикселах
`outerHeight`	Возвращает полную высоту окна в виде числа в пикселах
`screenX`	Возвращает горизонтальную координату левого верхнего угла окна относительно экрана компьютера в виде числа в пикселах
`screenY`	Возвращает вертикальную координату левого верхнего угла окна относительно экрана компьютера в виде числа в пикселах
`name`	Хранит имя, заданное для программно созданного окна, в виде строки (как открыть окно из сценария, мы узнаем позже). Если имя окна не было задано, хранит строку `"view"`
`opener`	Возвращает окно, из которого было программно открыто текущее окно. Для окон, открытых посетителем, возвращается `null`
`closed`	Возвращает `true`, если окно закрыто (неважно, программно или самим посетителем), и `false` в противном случае
`document`	Возвращает экземпляр объекта `HTMLDocument`, представляющий открытую в окне страницу (это свойство уже знакомо нам по предыдущим главам)

В следующем примере в переменных `rbX` и `rbY` окажутся, соответственно, горизонтальная и вертикальная координаты правого нижнего угла окна Web-обозревателя:

```
var rbX = window.screenX + window.outerWidth;
var rbY = window.screenY + window.outerHeight;
```

Методы `scrollTo` и `scrollBy` объекта `Window` выполняют прокрутку содержимого окна. Первый метод прокручивает содержимое в указанную позицию, второй — на заданную величину относительно текущей позиции:

```
<окно>.scrollTo|scrollBy(<по горизонтали>, <по вертикали>)
```

Координаты позиции, в которую нужно прокрутить окно, или величины прокрутки относительно текущей позиции по горизонтали и вертикали задаются, соответственно, первым и вторым аргументами в виде чисел в пикселах. В случае метода `scrollBy` положительные величины задают прокрутку вправо и вниз, отрицательные — влево и вверх. Результата эти методы не возвращают.

Вот пара примеров:

- прокручиваем содержимое окна до точки с координатами (100; 1000):

  ```
  window.scrollTo(100, 1000);
  ```

- после чего прокручиваем его содержимое вверх на 400 пикселов:

  ```
  window.scrollBy(0, -400);
  ```

Не принимающий аргументов и не возвращающий результата метод `print` открывает стандартное диалоговое окно печати, через которое посетитель может распечатать открытую в окне страницу.

Метод `open` объекта `Window` позволяет создать новое окно Web-обозревателя и сразу же открыть в нем какую-либо страницу:

```
<окно>.open([<интернет-адрес страницы>[, <имя окна>[, <параметры окна>[,
<заменить текущий интернет-адрес в истории?>]]]])
```

Интернет-адрес страницы, которая будет открыта в новом окне, задается в виде строки. Если он не указан, будет открыто пустое окно.

Имя окна может быть использовано в качестве цели гиперссылки, оно также задается в виде строки. Если имя не указано, новое окно не будет иметь имени.

Набор *параметров окна* задается в виде строки. Параметры указываются в формате *<имя параметра>=<значения параметра>* и разделяются запятыми. Вот доступные для указания параметры, поддерживаемые всеми Web-обозревателями:

- `width` — задает ширину клиентской области окна в пикселах;
- `height` — задает высоту клиентской области окна в пикселах;
- `left` — задает горизонтальную координату левого верхнего угла окна в пикселах;
- `top` — задает вертикальную координату левого верхнего угла окна в пикселах.

Если параметры не указаны, либо будет создана новая вкладка в текущем окне, либо новое окно будет иметь произвольные размеры и местоположение.

Если в последнем аргументе указать значение `true`, интернет-адрес страницы, открываемой в новом окне, заменит в списке истории Web-обозревателя интернет-адрес страницы, что была открыта в текущем окне. Если же указать `false` или вообще опустить последний аргумент, интернет-адрес открываемой в новом окне страницы будет, как обычно, добавлен в список истории.

Метод `open` возвращает экземпляр объекта `Window`, представляющий вновь созданное окно. Мы можем использовать его для управления окном, открытой в нем страницей и ее содержимым.

В следующем примере мы создаем новое окно без имени с размерами 640×480 пикселов и открываем в нем страницу `page2.html`:

```
window.open("page2.html", "", "width=640,height=480");
```

Листинг 20.1 показывает, как можно открыть в новом окне другую страницу и сразу же добавить в нее новый элемент — абзац.

Листинг 20.1

```
var oW = window.open("page3.html", "", "width=640,height=480");
var oP = document.createElement("p");
var oText = document.createTextNode("Этот абзац создан программно.");
oP.appendChild(oText);
oW.document.body.appendChild(oP);
```

> **Внимание!**
>
> Все методы объекта `Window`, рассматриваемые далее, действуют лишь на окна, созданные программно. Это сделано ради безопасности.

Методы `moveTo` и `moveBy` перемещают окно, соответственно, в указанную позицию экрана и на указанное расстояние от текущей позиции:

```
<окно>.moveTo|moveBy(<по горизонтали>, <по вертикали>)
```

Значения их аргументов указываются в том же виде, что и для методов `scrollTo` и `scrollBy`.

В следующем примере мы сдвигаем программно созданное окно на 200 пикселов вправо и на 50 пикселов вниз:

```
oW.moveBy(200, 50);
```

Методы `resizeTo` и `resizeBy` изменяют размеры окна, соответственно, до указанных величин или на указанные величины относительно текущих значений размеров. Они вызываются так же, как методы `moveTo` и `moveBy`.

В следующем примере мы задаем для программно созданного окна размеры 800×600 пикселов:

```
oW.resizeTo(800, 600);
```

И еще несколько полезных методов, не принимающих аргументов и не возвращающих результата:

- `focus` — делает окно активным;
- `blur` — делает окно неактивным;
- `close` — закрывает окно.

В следующем примере мы проверяем, не закрыто ли программно созданное окно посетителем, и, если это так, закрываем его:

```
if (!oW.closed) oW.close();
```

Осталось рассмотреть события, поддерживаемые окнами:

- `load` — возникает по окончании загрузки открытой в окне страницы, включая все привязанные к ней файлы (внешние таблицы стилей и файлы сценариев) и внедренные элементы (графические изображения, аудио- и видеоролики);
- `resize` — периодически возникает при изменении размеров окна;
- `scroll` — периодически возникает при прокрутке содержимого окна;
- `focus` — возникает, когда окно становится активным;
- `blur` — возникает, когда окно становится неактивным;
- `hashchange` — возникает при переходе на другой якорь;
- `beforeunload` — возникает непосредственно перед выгрузкой страницы, открытой в окне.

Если функция-обработчик этого события в процессе выполнения присвоит свойству `returnValue` экземпляра объекта, представляющего событие, какую-либо строку, перед выгрузкой страницы на экране появится окно-предупреждение, с текстом, взятым из упомянутого свойства, и кнопками **OK** и **Отмена**. Страница будет выгружена лишь в том случае, если посетитель нажмет в этом окне кнопку **OK**.

Если функция-обработчик этого события не будет выполнять описанные здесь действия, страница будет выгружена немедленно;

- `unload` — возникает при выгрузке страницы, открытой в окне, после события `beforeunload`;
- `error` — возникает, когда в коде сценария встречается ошибка. Функция-обработчик этого события может принимать следующие параметры:
  - текстовое сообщение об ошибке;
  - интернет-адрес файла со сценарием, в котором встретилась ошибка;
  - номер строки кода, в которой встретилась ошибка;
  - номер символа кода, в котором встретилась ошибка;
  - экземпляр объекта `Error`, хранящий сведения об ошибке. Он нам малополезен, так как поддерживает лишь свойство `message`, хранящее текстовое сообщение об ошибке, которое и так доступно из первого параметра функции-обработчика.

Обработчики к этим событиям можно привязать либо вызовом знакомого нам метода `addEventListener`, либо присвоив функцию-обработчик свойству с именем вида: `on`<*имя события*>.

Интернет-адрес текущей страницы

Интернет-адрес страницы, открытой в окне Web-обозревателя, представляется объектом `Location`. Его экземпляр можно получить из свойства `location` объекта `Window`.

Свойства объекта `Location` (табл. 20.2) хранят различные части интернет-адреса: обозначение протокола, имя хоста и проч. Их значения представляют собой строки, если иное не указано явно.

Таблица 20.2. *Свойства объекта* `Location`

Свойство	**Описание**
`href`	Хранит интернет-адрес целиком
`protocol`	Хранит обозначение протокола. Для страниц, загруженных с локального диска, хранит строку `"file:"`
`host`	Хранит комбинацию интернет-адреса Web-сервера (имени хоста) и номера порта (если он указан). Если страница хранится на локальном диске, хранит пустую строку

Таблица 20.2 *(окончание)*

Свойство	Описание
`hostname`	Хранит интернет-адрес Web-сервера (имени хоста). Если страница хранится на локальном диске, хранит пустую строку
`port`	Хранит номер порта или пустую строку, если он не указан
`pathname`	Хранит путь к запрашиваемому файлу или символ слеша, если таковой не указан
`search`	Хранит набор GET-параметров или пустую строку, если он отсутствует
`hash`	Хранит имя якоря вместе с символом # или пустую строку, если таковой отсутствует

В следующем примере мы открываем в текущем окне другую страницу:

```
window.location.href = "page2.html";
```

Методы `assign` и `replace` выполняют загрузку другой страницы, и вызовы этих методов могут служить альтернативой использованию свойства `href`. Различаются они тем, что метод `replace` заменяет текущую страницу новой в списке истории Web-обозревателя, а метод `assign` добавляет новую страницу в этот список:

```
<интернет-адрес>.assing|replace(<интернет-адрес открываемой страницы>)
```

Интернет-адрес открываемой страницы задается в виде строки. Результата оба этих метода не возвращают.

Пример:

```
window.location.assing("page2.html");
```

Метод `reload` выполняет перезагрузку текущей страницы:

```
<интернет-адрес>.reload([<загружать, минуя кэш?>])
```

Если передать этому методу единственным аргументом значение `false` или если вообще не указывать аргумент, страница, по возможности, будет перезагружена из кэша Web-обозревателя. Если же передать значение `true`, страница будет перезагружена непосредственно с Web-сервера, минуя кэш. Результата метод `reload` не возвращает.

Список истории Web-обозревателя

Список истории Web-обозревателя включает все страницы, открытые посетителем за определенный промежуток времени, и хранится на диске клиентского компьютера. Он представляется объектом `History`, экземпляр которого доступен через свойство `history` объекта `Window`.

Объект `History` представляет собой коллекцию, следовательно, как все коллекции, поддерживает свойство `length`, возвращающее количество элементов в ней.

Не принимающие аргументов и не возвращающие результата методы `back` и `forward` выполняют переход, соответственно, на предыдущую и следующую страницы в списке истории.

В следующем примере мы возвращаемся на предыдущую страницу:

```
window.history.back();
```

Метод `go` позволяет прямо указать страницу из списка истории, на которую требуется перейти:

<список истории>`.go(`*<количество пропускаемых позиций>*`|`*<интернет-адрес>*`)`

Нужную страницу мы можем указать двумя способами:

- в виде числа, задающего количество позиций списка истории, на которое отстоит требуемая позиция от текущей: положительное число указывает, что требуется сместиться вперед, отрицательная — назад;
- в виде заданного строкой интернет-адреса.

Результата метод `go` не возвращает.

Вот пара примеров:

- переходим на страницу, расположенную через одну позицию от текущей, если считать «назад»:

  ```
  window.history.go(-2);
  ```

- переходим на страницу с указанным интернет-адресом:

  ```
  window.history.go("http://www.somesite.ru/page3.html");
  ```

Параметры экрана

Web-обозреватель позволяет получить некоторые параметры экрана клиентского компьютера. Они могут пригодиться нам, скажем, при позиционировании и задании размеров программно созданных окон.

Параметры экрана хранит объект `Screen`, экземпляр которого мы без проблем получим из свойства `screen` объекта `Window`.

Объект `Screen` поддерживает лишь свойства, доступные только для чтения (что и понятно, ведь мы не можем изменить параметры экрана программно). Все они приведены в табл. 20.3.

Таблица 20.3. *Свойства объекта* `Screen`

Свойство	**Описание**
`availWidth`	Возвращает ширину доступной области экрана (без учета панели задач и инструментальных панелей) в виде числа в пикселах
`availHeight`	Возвращает высоту доступной области экрана (без учета панели задач и инструментальных панелей) в виде числа в пикселах

Таблица 20.3 *(окончание)*

Свойство	Описание
`width`	Возвращает полную ширину экрана (с учетом панели задач и инструментальных панелей) в виде числа в пикселах
`height`	Возвращает полную высоту экрана (с учетом панели задач и инструментальных панелей) в виде числа в пикселах
`colorDepth`	Возвращает глубину цвета (количество битов, которыми кодируется цвет) в виде числа в битах на пиксел. Для экранных подсистем настольных компьютеров возвращает 24 (24-разрядный цвет), величина 1 означает, что установлен монохромный экран.

В следующем примере мы программно создаем окно, занимающее всю доступную область экрана:

```
window.open("page2.html", "", "width=" + window.screen.availWidth +
",height=" + window.screen.availHeight);
```

Сведения о Web-обозревателе

Наконец, мы можем получить сведения о самом Web-обозревателе. Для этого служит объект `Navigator`, экземпляр которого хранится в свойстве `navigator` объекта `Window`.

У этого объекта нас интересуют лишь свойства (табл. 20.4). Все они доступны только для чтения и возвращают строковые значения, если иное не указано явно.

Таблица 20.4. *Свойства объекта* `Navigator`

Свойство	Описание
`appName`	Возвращает строку `"Netscape"` для Internet Explorer 11, Firefox, Chrome и Safari, `"Microsoft Internet Explorer"` для Internet Explorer 10 и более старых версий, `"Opera"` для Opera
`appVersion`	Возвращает длинную строку, содержащую все сведения о Web-обозревателе и операционной системе клиентского компьютера
`cookieEnabled`	Возвращает `true`, если Web-обозревателю разрешено принимать cookie, и `false` в противном случае
`onLine`	Возвращает `true`, если Web-обозреватель подключен к сети, и `false`, если он работает в автономном режиме
`platform`	Возвращает строковое обозначение операционной системы, на которой работает Web-обозреватель: `"Win32"`, `"WinCE"`, `"MacIntel"`, `"Linux i686"`, `"Linux armv7l"` и др.
`userAgent`	Возвращает длинную строку, содержащую все сведения о Web-обозревателе и схожую с той, что возвращается свойством `appVersion`

Стандартные диалоговые окна и сообщения

Web-обозреватель предоставляет нам возможность вывести стандартные предупреждения, сообщающие о чем-либо посетителю или получающие от него разрешение на выполнение какой-либо операции. Мы можем также вывести стандартное диалоговое окно запроса у посетителя какого-либо значения (но эта возможность пригодится разве что для отладки сценариев).

Для вывода обычных окон-сообщений с текстом и кнопкой **OK** служит метод `alert`:

```
<окно>.alert(<сообщение>)
```

Сообщение задается в виде строки. Сам метод не возвращает результата.

Пример:

```
window.alert("Проверка!");
```

Вывести сообщение с текстом и кнопками **OK** и **Отмена** можно вызовом метода `confirm`. Формат его вызова такой же, как и у метода `alert`.

Если посетитель нажал кнопку **OK** появившегося на экране сообщения, метод `confirm` вернет в качестве результата `true`. Если же посетитель нажал кнопку **Отмена** или просто закрыл сообщение, щелкнув по стандартной кнопке закрытия в его заголовке, метод вернет `false`.

Листинг 20.2 показывает пример реализации перехода по гиперссылке только в том случае, если посетитель щелкнет кнопку **OK** в появившемся на экране окне-предупреждении.

Листинг 20.2

```
<a id="link" href=". . .">Перейти на другой сайт</a>
. . .
var oLink = document.getElementById("link");
oLink.addEventListener("click", function(evt) {
  if (!window.confirm("Выполнить переход на другой сайт?"))
    evt.preventDefault();
});
```

Чтобы отказаться от перехода по гиперссылке, мы просто отменяем обработчик события `click` по умолчанию, вызвав метод `preventDefault`.

Для вывода стандартного диалогового окна для запроса значения служит метод `prompt`:

```
<окно>.prompt(<пояснение>[, <значение по умолчанию>])
```

Пояснение описывает данные, которые посетителю нужно ввести. *Значение по умолчанию*, если оно указано, будет подставлено в поле ввода диалогового окна изначально, сразу после его открытия. Обе величины задаются в виде строк.

Если посетитель занес какое-либо значение в поле ввода диалогового окна и нажал кнопку **OK**, метод `prompt` вернет само введенное значение. Если же посетитель нажал кнопку **Отмена** или закрыл окно щелчком на кнопке закрытия в его заголовке, будет возвращено значение `null`.

Листинг 20.3 показывает код, который запрашивает у посетителя его имя и выводит на экран приветствие.

Листинг 20.3

```
<div id="output"></div>
. . .
var oOutput = document.getElementById("output");
var sName = window.prompt("Введите ваше имя");
if (sName)
  oOutput.textContent = "Здравствуйте, " + sName + "!"
```

В этом фрагменте кода мы использовали прием, связанный с особенностями преобразования типов в языке JavaScript (за подробностями — к *главе 16*). Дело в том, что непустая строка всегда преобразуется в `true`, а пустая и `null` — в `false`. Поэтому, если мы поставим в качестве условия в условное выражение результат, возвращенный методом `prompt`, условие выполнится в случае, когда этот результат представляет собой непустую строку, и не выполнится, если этим результатом является `null`.

Таймеры

Напоследок мы познакомимся с инструментами, позволяющими выполнять какой-либо код каждый раз по истечении определенного промежутка времени — *таймерами*. Для работы с ними применяются четыре особых метода объекта `Window`.

Прежде всего нужно заметить, что поддерживаются таймеры двух типов. Сейчас мы их рассмотрим.

Таймеры первого типа носят название *повторяющихся*, или *интервалов*. Они выполняют указанный код, оформленный в виде функции, снова и снова, пока страница не будет выгружена или пока сам таймер не будет удален.

Метод `setInterval` создает повторяющийся таймер:

```
<окно>.setInterval(<функция>, <значение интервала>[,
<аргументы, передаваемые функции>])
```

`Функция`, которая будет выполняться каждый раз по истечении интервала, может быть как именованной, так и анонимной. *`Значение интервала`* задается в виде числа и измеряется в миллисекундах.

Метод `setInterval` возвращает в качестве результата число, обозначающее созданный интервал, — *идентификатор интервала*. Оно пригодится нам впоследствии, если мы захотим удалить этот интервал.

Как уже говорилось, функция, указанная в вызове метода `setInterval`, будет вызываться до тех пор, пока посетитель не закроет страницу или пока мы сами не удалим интервал программно. Для удаления интервала применяется метод `clearInterval`:

<окно>`.clearInterval(`*<идентификатор интервала>*`)`

Значения он не возвращает.

В листинге 20.4 приведен пример кода, плавно увеличивающего размер шрифта текста в блоке `animated` от 10 до 48 пунктов.

Листинг 20.4

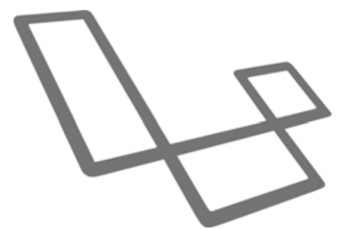
```
<div id="animated">Анимированный текст.</div>
. . .
function animStep() {
  fsCurrent += fsDelta;
  if (fsCurrent > fsMax) {
    window.clearInterval(idInterval);
  } else {
    oAnimated.style.fontSize = fsCurrent + "pt";
  }
}
var oAnimated = document.getElementById("animated");
var fsCurrent = 10, fsMax = 48, fsDelta = 2;
var idInterval = window.setInterval(animStep, 100);
```

Таймеры второго типа, напротив, выполняют указанную функцию всего один раз, после чего сами удаляются. Это *неповторяющиеся таймеры*, или *тайм-ауты*.

Создать тайм-аут можно вызовом метода `setTimeout`, формат вызова которого совпадает с таковым у метода `setInterval` и который возвращает в качестве результата *идентификатор тайм-аута*. А удалить его можно вызовом метода `clearTimeout`, который вызывается так же, как метод `clearInterval`.

Пример:

```
<p>Через три секунды вы будете перенаправлены на другую страницу.</p>
. . .
window.setTimeout(function() {
  window.location.assign("http://www.othersite.ru/");
}, 3000);
```



ГЛАВА 21

Работа с локальными файлами. Регулярные выражения

Одно из весьма полезных нововведений, совсем недавно появившихся в составе DOM, — инструменты для работы с файлами, хранящимися на локальном компьютере. Так что теперь, реализуя, скажем, выгрузку графических файлов на сервер, мы можем реализовать их предварительный просмотр.

Еще язык JavaScript поддерживает регулярные выражения — исключительно мощное средство для выполнения поиска и замены фрагментов текста. Особенно оно полезно, если нам требуется найти в тексте не точно указанную подстроку, а фрагмент, удовлетворяющий определенным условиям, — например, являющийся адресом электронной почты.

Работа с локальными файлами

Перед тем как начать рассказ о средствах для работы с локальными файлами, следует сделать два важных замечания:

- мы имеем возможность получить доступ только к тем файлам, на которые нам укажет сам посетитель, выбрав их с помощью поля ввода файла (этот элемент управления был описан в *главе 6*);
- мы можем лишь загрузить выбранный посетителем файл (файлы). Создавать, сохранять, переименовывать, перемещать и удалять файлы мы не можем.

Эти ограничения реализованы ради безопасности.

Получение выбранных файлов и сведений о них

Итак, посетитель выбрал в поле ввода файла какой-либо файл или даже сразу несколько файлов. Как нам получить доступ к ним? Через свойство `files` объекта, представляющего поле ввода файла. Оно возвращает экземпляр объекта-коллекции `FileList`, который как раз и хранит список выбранных файлов.

> **ВНИМАНИЕ!**
>
> Поле ввода файла также поддерживает свойство `value`, которое у разных Web-обозревателей возвращает либо поддельный путь к выбранному файлу (в котором достоверно только имя файла), либо только имя этого файла — в виде строки. Понятно, что пользы от этого свойства немного.

Каждый элемент объекта-коллекции `FileList` является экземпляром объекта `File`, представляющим отдельный файл. Мы можем получить доступ к отдельным файлам из списка таким же образом, каким ранее получали доступ к элементам любой коллекции.

Объект `File` поддерживает следующие свойства:

- `name` — возвращает только имя файла без пути (это, опять же, сделано в целях безопасности);
- `size` — возвращает размер файла в виде числа в байтах;
- `lastModifiedDate` — возвращает дату последнего изменения файла в виде экземпляра объекта `Date`;
- `type` — возвращает MIME-тип файла или пустую строку, если тип опознать не удается.

Листинг 21.1 показывает пример формы с полем ввода файла. После нажатия кнопки **Вывести сведения** на экране появятся сведения о выбранном посетителем файле.

Листинг 21.1

```
<form>
  <p>Выберите файл: <input id="txtFile" type="file"></p>
  <p><input type="submit" value="Вывести сведения"></p>
</form>
<p id="output"></p>
. . .
var txtFile = document.getElementById("txtFile");
var output = document.getElementById("output");
txtFile.form.addEventListener("submit", function(evt) {
  var f = txtFile.files[0];
  output.innerHTML = "Файл \"" + f.name + "\"<br>" +
  "Размер, байт: " + f.size + "<br>" +
  "Дата изменения: " + f.lastModifiedDate.toLocaleDateString() + "<br>" +
  "MIME-тип: " + f.type;
  evt.preventDefault();
});
```

Не принимающий аргументов метод `toLocaleDateString` объекта `Date` возвращает строку с датой, отформатированной согласно текущим языковым настройкам операционной системы.

Загрузка выбранных файлов

Чтобы загрузить выбранные посетителем файлы и таким образом получить их содержимое, применяется особый объект `FileReader`.

Экземпляр этого объекта создается с помощью оператора `new`. Никаких аргументов при этом не передается. Пример:

```
var fr = new FileReader();
```

Метод `readAsText` этого объекта применяется для загрузки текстовых файлов:

```
<экземпляр объекта FileReader>.readAsText(<файл>[, <кодировка>])
```

Файл задается в виде экземпляра объекта `File`. Вторым аргументом может быть указана строка с обозначением кодировки, в которой сохранен считываемый файл, если он не указан, используется кодировка UTF-8. Результата этот метод не возвращает.

Метод `readAsDataURL` того же объекта поможет, если требуется загрузить нетекстовый файл (изображение, аудио- или видеоролик):

```
<экземпляр объекта FileReader>.readAsDataURL(<файл>)
```

Файл также указывается в виде экземпляра объекта `File`. Результата этот метод также не возвращает.

Загруженный с применением метода `readAsDataURL` файл будет представлен в виде *Data URL* — особого интернет-адреса, который не указывает на местоположение файла, а хранит само его содержимое. Data URL может быть использован вместо обычного интернет-адреса для вывода загруженного файла на экран.

Особенностью обоих указанных методов является то, что они лишь запускают загрузку файла и сразу же после этого завершают свою работу, не дожидаясь, когда она закончится. Чтобы отследить момент окончания загрузки файла (или возникновения ошибки в процессе его загрузки), следует обрабатывать события объекта `FileReader`, а содержимое загруженного файла — получать через свойства того же объекта.

Также нужно упомянуть о методе `abort` объекта `FileReader`. Он прерывает загрузку файла, не принимает аргументов и не возвращает результата.

Теперь рассмотрим свойства этого объекта. Их немного:

- `result` — возвращает считанное содержимое файла в виде строки. Значение этого свойства имеет смысл только в случае успешной загрузки файла;
- `error` — возвращает экземпляр объекта `DOMError`, хранящий сведения о возникшей в процессе загрузки файла ошибке. У этого объекта нас интересует свойство `name`, которое хранит строку с названием ошибки;
- `readyState` — возвращает обозначение состояния загрузки файла в виде числа:
  - 0 — загрузка еще не началась;
  - 1 — загрузка в данный момент идет;
  - 2 — загрузка завершена либо удачно, либо неудачно.

Осталось описать события объекта `FileReader`:

- `loadstart` — возникает в начале загрузки файла;
- `progress` — периодически возникает в процессе загрузки файла и может использоваться для вывода индикатора процесса его считывания;
- `load` — возникает после успешной загрузки файла;
- `error` — возникает при неудачной загрузке файла вследствие ошибки;
- `abort` — возникает при прерывании загрузки файла вызовом метода `abort`;
- `loadend` — возникает в самом конце загрузки файла, после события `load`, `error` или `abort`.

События `load`, `error` и `abort` представляются объектом `Event` (см. *главу 18*) и, следовательно, не предоставляют каких-либо дополнительных сведений. А вот события `loadstart`, `progress` и `loadend` представляются описанным в *главе 19* объектом `ProgressEvent`, который позволяет узнать как общий размер файла, так и размер уже загруженной его части.

Листинг 21.2 показывает пример кода, загружающего текстовый файл и выводящего на экран его содержимое.

Листинг 21.2

```
<form>
  <p>Выберите файл: <input id="txtFile" type="file" accept=".txt"></p>
  <p><input type="submit" value="Вывести файл"></p>
</form>
<div id="output"></div>
. . .
var txtFile = document.getElementById("txtFile");
var output = document.getElementById("output");
txtFile.form.addEventListener("submit", function(evt) {
  var f = txtFile.files[0];
  var fr = new FileReader();
  fr.addEventListener("load", function(evt) {
    output.textContent = evt.target.result;
  });
  fr.readAsText(f);
  evt.preventDefault();
});
```

А листинг 21.3 представляет пример реализации простого просмотрщика графических файлов, который может вывести на экран сразу несколько изображений в виде своего рода фотогалереи (рис. 21.1).

Листинг 21.3

```
/* Блоки, в которые будут помещены отдельные изображения, делаем
плавающими, задаем для них размеры, внешние отступы, чтобы отодвинуть их
друг от друга, и выравнивание содержимого — самих изображений —
по середине */
#output div     { float: left;
                  width: 150px;
                  height: 100px;
                  margin: 10px;
                  text-align: center; }
/* Для самих изображений указываем максимальные размеры, равные размерам
блоков, в которые они помещены, чтобы первые полностью помещались в
последние */
#output div img { max-width: 150px;
                  max-height: 100px; }
. . .
<!-- Форма для выбора файлов -->
<form>
  <p>Выберите файлы: <input id="txtFile" type="file" accept="image/*"
  multiple></p>
  <p><input type="submit" value="Вывести файлы"></p>
</form>
<!-- Элемент-фотогалерея, в котором будут выводиться изображения -->
<div id="output"></div>
. . .
// Функция — обработчик события load экземпляра объекта FileReader,
// загружающего очередной графический файл. Она создаст блок и
// изображение (тег img), в последнем выведет содержимое загруженного
// файла, добавит изображение в блок, а блок — в элемент-фотогалерею
function loadImage(evt) {
  var div = document.createElement("div");
  var img = document.createElement("img");
  img.src = evt.target.result;
  div.appendChild(img);
  output.appendChild(div);
}

// Этот код, собственно, выполнит загрузку файлов
var txtFile = document.getElementById("txtFile");
var output = document.getElementById("output");
txtFile.form.addEventListener("submit", function(evt) {
  var fs = txtFile.files;
  for (var i = 0; i < fs.length; i++) {
    var fr = new FileReader();
    fr.addEventListener("load", loadImage);
```

```
    fr.readAsDataURL(fs[i]);
  }
  evt.preventDefault();
});
```

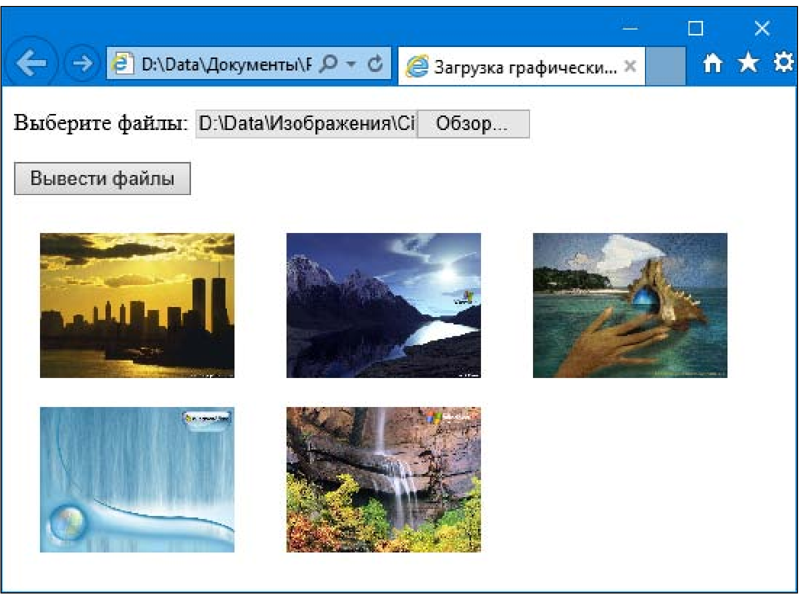


Рис. 21.1. Просмотрщик графических файлов,
код которого показан в листинге 21.3

Регулярные выражения

Регулярное выражение можно рассматривать как шаблон, которому должен соответствовать искомый фрагмент (подстрока) текста. При выполнении поиска в строке будут отбираться только те подстроки, что соответствуют этому шаблону. А если регулярное выражение указано в теге поля ввода (с помощью атрибута `pattern`, см. *главу 6*), занесенное в это поле значение будет сравниваться с ним.

Написание регулярных выражений. Литералы и группы

Регулярное выражение может содержать как обычные символы, которые обязательно должны присутствовать в искомом фрагменте, так и специальные символы — *литералы*. Литералы либо задают определенную группу символов, которая должна входить в состав фрагмента, либо указывают специальные условия поиска.

Все литералы, которые мы можем использовать в регулярных выражениях, приведены в табл. 21.1.

Таблица 21.1. *Литералы регулярных выражений*

Литерал	Описание
.	Любой символ, за исключением перевода строки
\w	Алфавитно-цифровой символ (буква, цифра или подчеркивание)
\W	Не алфавитно-цифровой символ
\d	Цифра
\D	Не цифра
\s	Пробельный символ (пробел, табуляция, возврат каретки, перевод строки или прогон листа)
\S	Не пробельный символ
\r	Возврат каретки
\n	Перевод строки
\t	Табуляция
\f	Прогон листа
\u<*код*>	Символ с *кодом*, заданным в кодировке Unicode
\b	Начало или конец слова (таковым считается любой пробельный символ или знак препинания)
\B	Не начало и не конец слова
^	Начало строки
$	Конец строки
[<*набор*>]	Любой символ из *набора*
[^<*набор*>]	Ни один символ из *набора*
[<*A*>-<*B*>]	Любой символ из диапазона от символа *A* до символа *B*
[^<*A*>-<*B*>]	Ни один символ из диапазона от символа *A* до символа *B*
(<*A*>|<*B*>)	Либо подстрока *A*, либо подстрока *B*
*	Предыдущий символ может присутствовать произвольное количество раз, а может и не присутствовать вообще
+	Предыдущий символ должен присутствовать в строке как минимум один раз
?	Предыдущий символ должен либо не присутствовать в строке, либо присутствовать один раз
{<*n*>}	Предыдущий символ должен присутствовать в строке строго *n* раз
{<*n*>,}	Предыдущий символ должен присутствовать в строке, как минимум, *n* раз
{<*m*>,<*n*>}	Предыдущий символ должен присутствовать в строке от *m* до *n* раз
(?=<подстрока>)	За предыдущим символом должна следовать подстрока, отделенная от него произвольным количеством символов
(?!<подстрока>)	За предыдущим символом не должна следовать подстрока

Если нам потребуется указать в регулярном выражении литерал в качестве обычного символа, мы предварим его символом обратного слеша (\).

Несколько примеров регулярных выражений можно увидеть в табл. 21.2.

Таблица 21.2. *Примеры регулярных выражений*

Регулярное выражение	Описание
Java.+	Совпадает с подстрокой, начинающейся на "Java": "JavaScript", "JavaScript2", "Javaaaaaa" и т. п.
HTML?	Совпадает со строками "HTM" и "HTML"
\d{2,4}	Совпадает с числом с количеством цифр от 2 до 4: 12, 123, 7890
[a-z]{3}	Совпадает с тремя строчными латинскими буквами: "htm", "txt", "css"
(css|js)	Совпадает с подстрокой "css" и подстрокой "js"
<h\d>.*<\/h\d>	Совпадает с тегом заголовка и его содержимым: " Заголовок ", " Заголовок ". (Обратим внимание, как мы предварили символом обратного слеша знак /, чтобы указать его в регулярном выражении в качестве обычного символа.)
[^\.,!\?:;]	Совпадает с любым символом, не являющимся точкой, запятой, восклицательным или вопросительным знаком, двоеточием и точкой с запятой. (Здесь мы также предварили символами обратного слеша точку и восклицательный знак, чтобы указать их в качестве обычных символов.)
[a-z]{8}(?=\.txt)	Совпадает с восемью прописными латинскими буквами, за которыми следует подстрока ".txt"

В регулярных выражениях мы можем создавать *группы*. Группа позволяет выделить фрагмент внутри уже найденной подстроки, возможно, задать на его основе какое-либо дополнительное условие поиска и, также возможно, сохранить найденный фрагмент для дальнейшего применения. Группы всегда обрабатываются как независимые части регулярного выражения, и мы можем это использовать.

Вот литералы, предназначенные для работы с группами:

- (*<фрагмент>*) — выделяет *фрагмент* внутри найденной подстроки и сохраняет его для дальнейшего использования, формируя тем самым группу;
- *<номер>* — совпадает с уже существующей группой, имеющей заданный порядковый *номер* (номера группы считаются от начала регулярного выражения и начинаются с единицы).

Давайте рассмотрим несколько примеров использования групп регулярных выражений (табл. 21.3).

По умолчанию поиск с применением регулярного выражения выполняется так, чтобы захватить самую длинную подстроку, которая совпадает с ним (так называе-

Таблица 21.3. *Примеры использования групп регулярных выражений*

Регулярное выражение	Описание
`Глава (\d+)`	Совпадает с подстроками вида "`Глава` *`<числовой номер>`*" ("`Глава 5`", "`Глава 27`") и формирует группу на основе найденного *номера*, тем самым сохраняя его для дальнейшего использования
`search=(.+)`	Совпадает с подстрокой вида "`search=`*`<строка>`*" и сохраняет найденную *`строку`* для дальнейшего использования
`(Глава |Гл\.)(\d+)`	Совпадает с подстроками вида "`Глава |Гл.`*`<числовой номер>`*" ("`Глава 5`", "`Глава 57`", "`Гл.3`"). Слово "`Глава`" или "`Гл.`" и *номер* сохраняются для дальнейшего использования
`<h(\d)>(.*)<\/h\1>`	Уточненный пример поиска тегов заголовков. Совпадает лишь в том случае, если закрывающий тег соответствует открывающему. Помимо этого, сохраняет в отдельной группе содержимое тега

мый «жадный» поиск). Однако мы можем указать использовать «экономный» поиск, при котором будет захвачена, по возможности, самая короткая подходящая подстрока.

Для этого используется специальный литерал `?`. Он ставится за литералами `*`, `+`, `?` или `{. . .}` и указывает искать самую короткую последовательность символов, задаваемую предыдущими литералами регулярного выражения.

Для примера возьмем строку "`(HTML) и (CSS)`". Если мы используем регулярное выражение `\(.*\)` (оно совпадает с любым количеством символов, заключенных в круглые скобки), то получим подстроку "`(HTML) и (CSS)`". Как видим, она включает все символы между первой открывающей и последней закрывающей круглыми скобками, то есть налицо «жадный» поиск.

Теперь попробуем в деле регулярное выражение `\(.*?\)` (обратим внимание на литерал `?` после литерала `*`). В этом случае будет получена подстрока "`(HTML)`" — самая короткая из всех заключенных в скобки, то есть был выполнен «экономный» поиск.

Регулярные выражения — исключительно мощный, но и весьма сложный инструмент. Поэтому при их написании нам не обойтись без тестирующего средства, позволяющего проверить в работе как регулярное выражение целиком, так и его отдельные части. Таким средством может оказаться популярный онлайновый сервис тестирования регулярных выражений RegExr (**http://www.regexr.com/**).

Работа с регулярными выражениями

В JavaScript регулярные выражения представляются объектом `RegExp`. Создать экземпляр этого объекта можно, записав регулярное выражение в двух форматах:

- `/`*`<регулярное выражение>`*`/[`*`<модификаторы>`*`]` — то есть заключив само *`регулярное выражение`* в символы слеша, а его *`модификаторы`* поставив после заключительного символа слеша (о модификаторах будет рассказано чуть позже);

- `new RegExp(<регулярное выражение>[, <модификаторы>])` — применив оператор `new`. И *регулярное выражение*, и *модификаторы* указываются в виде строк.

Модификаторы позволяют указать некоторые условия поиска. Всего их три:

- `g` — если указан, будет вестись поиск всех подстрок, совпадающих с регулярным выражением (глобальный поиск), если не указан — только первой подходящей подстроки;
- `i` — если указан, регистр символов при поиске учитываться не будет, если не указан — будет;
- `m` — если указан, литерал `^` будет обозначать начало каждой строки текста, а литерал `$` — конец каждой строки. (Строки разделяются последовательностью символов возврата каретки и перевода строки: `"\r\n"`.) Если не указан, литералы `^` и `$` будут обозначать начало и конец всего текста соответственно. Это так называемый *многострочный* поиск.

Мы можем комбинировать модификаторы, чтобы объединить их действие, просто указывая их друг за другом без всяких разделителей.

Вот пара примеров:

- создаем регулярное выражение для выделения заданной посетителем подстроки поиска из интернет-адреса и задаем для него игнорирование регистра символов:

  ```
  var oR1 = /search=(.+)/i;
  ```

- создаем регулярное выражение для поиска номеров глав, указав глобальный и многострочный поиск:

  ```
  var oR2 = new RegExp("Глава (\d+)", "gm");
  ```

Выполнять поиск с помощью регулярных выражений можно воспользовавшись методами объекта `RegExp`. Их два:

- метод `test` выполняет проверку, совпадает ли строка с регулярным выражением, у которого был вызван этот метод:

  ```
  <регулярное выражение>.test(<строка>)
  ```

  Метод возвращает `true`, если строка совпадает с регулярным выражением, и `false` в противном случае.

  В следующем примере в переменной `f1` окажется значение `true`, а в переменной `f2` — `false`:

  ```
  var r = /Java.+/;
  var f1 = r.test("JavaScript");
  var f2 = r.test("Java");
  ```

- метод `exec` ищет в строке подстроку, совпадающую с регулярным выражением, у которого он был вызван:

  ```
  <регулярное выражение>.exec(<строка>)
  ```

  Результат возвращается в виде массива. Его первый элемент содержит найденную подстроку, второй и последующий элементы — подстроки, совпадающие

с находящимися в составе регулярного выражения группами (если, конечно, они там есть). Если же поиск не увенчается успехом, возвращается `null`.

Массив, возвращенный методом `exec`, поддерживает дополнительные свойства `input` (возвращает саму строку, переданную методу в качестве аргумента) и `index` (возвращает номер символа, с которого начинается найденная подстрока).

Пример использования этого метода приведен в листинге 21.4.

Листинг 21.4

```
var r = /Java(.+)/;
var a = r.exec("Язык JavaScript");
var s1 = a[0];
var s2 = a[1];
var n = a.index;
```

В переменной `s1` окажется строка `"JavaScript"`, в переменной `s2` — `"Script"`, а в переменной `n` — число `5`.

Еще объект `RegExp` поддерживает пять свойств:

- `lastIndex` — хранит номер символа в строке, с которого начнется поиск (значение по умолчанию — 0, то есть поиск будет выполняться с первого символа строки). Это свойство принимается во внимание, только если включен глобальный поиск (задан модификатор `g`), и может быть использовано, чтобы выполнить поиск не с начала строки, а с другого ее места;
- `ignoreCase` — возвращает `true`, если включен поиск без учета регистра символов, и `false` в противном случае;
- `multiline` — возвращает `true`, если включен многострочный поиск, и `false` в противном случае;
- `global` — возвращает `true`, если включен глобальный поиск, и `false` в противном случае;
- `source` — возвращает само регулярное выражение в виде строки.

Объект `String`, представляющий строку, также поддерживает средства для поиска и даже замены с применением регулярных выражений в виде трех методов:

- метод `search` ищет в строке, у которой вызван, первую подстроку, совпадающую с регулярным выражением:

  <строка, в которой выполняется поиск>`.search(`*<регулярное выражение>*`)`

  Он возвращает либо номер символа, с которого начинается найденная подстрока, либо –1, если поиск не увенчался успехом.

  В следующем примере в переменной `n` окажется число 6 — номер символа, с которого начинается найденная подстрока:

  ```
  var r = /\d+/;
  var s = "Глава 21. Работа с локальными файлами. Регулярные выражения";
  var n = s.search(r);
  ```

- метод `match` делает то же самое, но дополнительно позволяет получить подстроки, совпадающие с группами, созданными в регулярном выражении. Он вызывается так же, как метод `search`.

  В качестве результата, если поиск оказался успешным, он возвращает массив:

  - в случае выполнения обычного, «неглобального» поиска:
    - первый элемент этого массива будет содержать первую подстроку, совпадающую с регулярным выражением;
    - второй и последующие элементы массива будут содержать подстроки, совпадающие с группами регулярного выражения;
  - в случае выполнения глобального поиска массив будет содержать все подстроки, совпадающие с регулярным выражением.

  Если поиск не увенчался успехом, возвращается `null`.

  Пример использования метода `match` можно увидеть в листинге 21.5.

Листинг 21.5

```
var r = /Глава (\d+)"/i;
var s = "Глава 21. Работа с локальными файлами. Регулярные выражения";
var a = s.search(r);
var n = parseInt(arr[1]);
```

Получаем в переменной `n` число 21 — номер главы, преобразованный в числовой формат.

- метод `replace` заменяет найденную подстроку другой:

  ```
  <строка, в которой выполняется поиск>.replace(<регулярное выражение>,
  <заменяющая подстрока>)
  ```

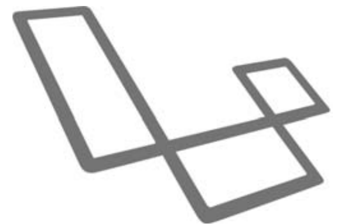
  Заменяющая подстрока задается в строковом формате. В ней мы можем использовать следующие литералы:

  - `$&` — найденная подстрока полностью;
  - `$<номер>` — подстрока, совпадающая с группой регулярного выражения с указанным порядковым *номером*;
  - `` $` `` — часть строки, предшествующая найденной подстроке;
  - `$'` — часть строки, следующая за найденной подстрокой;
  - `$$` — символ `$`.

  Метод `replace` возвращает новую строку со всеми выполненными заменами.

  В следующем примере мы получаем в переменной `s` строку `"Гл.21. Работа с локальными файлами. Регулярные выражения"`:

  ```
  var r = /Глава (\d+)"/i;
  var s = "Глава 21. Работа с локальными файлами. Регулярные выражения";
  s = s.replace(r, "Гл.$1");
  ```



ГЛАВА 22

AJAX

Еще в *главе 19* мы узнали, что можем программно указать графический файл, содержимое которого будет выводиться в элементе графического изображения — теге `<img>`. Для этого достаточно присвоить строку с интернет-адресом нового файла свойству `src` экземпляра объекта, представляющего этот элемент:

```
<img id="someimage" src="picture1.jpg">
. . .
var oSomeImage = document.getElementById("someimage");
oSomeImage.src = "picture2.jpg";
```

Как только этот код выполнится, Web-обозреватель сам загрузит файл `picture2.jpg` и выведет его в том месте страницы, где ранее выводился файл `picture1.jpg`.

А можем ли мы загрузить таким же образом не графический или, скажем, файл с аудио- или видеороликом, а файл, хранящий список статей? А потом, загрузив его, вывести на страницу в виде набора абзацев или заголовков с гиперссылками, указывающими на соответствующую статью?

Введение в AJAX

Конечно, можем. Технология, предназначенная для загрузки сторонних данных после завершения загрузки самой страницы, существует уже около 15 лет. И носит обобщенное название *AJAX* (Asynchronous JavaScript and XML, асинхронный JavaScript и XML).

Работает она следующим образом. С помощью особого сценария мы инициируем загрузку файла с Web-сервера или от серверной программы, возможно, передав при этом ей какие-либо параметры. А как только файл будет получен, мы прочитаем его содержимое и выведем его на экран.

Данные какого рода мы можем загружать таким образом? Трех различных типов, указанных далее:

- фрагменты HTML-кода, которые мы можем просто вывести на экран, вставив их в специально отведенный для этого элемент страницы;

- данные, закодированные с помощью языка *XML* (eXtensible Markup Language, расширяемый язык разметки). Этот язык применяется для кодирования структурированных данных любого рода. Понятно, что просто вывести эти данные на экран не получится, и нам потребуется раскодировать их и преобразовать в подходящий для вывода в составе страницы формат — HTML-код;
- данные, закодированные в формате JSON (о нем речь пойдет чуть позже). Их также потребуется раскодировать и преобразовать в HTML-код.

Как видим, технология AJAX, хоть и включает в свое полное название аббревиатуру XML, отнюдь не ограничивает нас в формате принимаемых с ее помощью данных.

Для загрузки данных всех трех указанных форматов мы воспользуемся одним и тем же набором программных инструментов. Разница будет заключаться лишь в способах обработки полученных данных перед выводом.

Программная реализация AJAX

Не будем тратить время. Давайте сразу же начнем разговор о программных инструментах, которыми в современных Web-обозревателях реализуется технология AJAX.

Объект *XMLHttpRequest*

Загрузкой данных по технологии AJAX в Web-обозревателях «заведует» специально отведенный для этой задачи объект `XMLHttpRequest`. Отметим, что он только загружает данные — их обработкой и выводом мы должны заниматься самостоятельно.

Первый шаг, который нам нужно предпринять для получения данных AJAX, — создание экземпляра этого объекта. Выполняется это знакомым нам по *главе 16* оператором `new`, никакие аргументы при этом не указываются:

```
var oAJAX = new XMLHttpRequest();
```

Отправка запроса

Следующий шаг — отправка запроса Web-серверу на получение данных AJAX. Мы укажем метод отправки данных, интернет-адрес получаемого файла или серверной программы, которая предоставит нам данные, и признак, указывающий, является отправляемый нами запрос синхронным или асинхронным (о разнице между ними мы поговорим позже).

Все эти действия выполняет метод `open` объекта `XMLHttpRequest`:

```
<экземпляр объекта XMLHttpRequest>.open(<метод отправки данных>,
<интернет-адрес>, <асинхронный запрос?>)
```

`Метод отправки данных` указывается в виде строки `"GET"` или `"POST"`. *`Интернет-адрес`*, с которого запрашиваются данные, также указывается как строка. Если третьим

аргументом передано значение true, будет выполнен асинхронный запрос, если false — синхронный. Результата метод open не возвращает.

Если мы отправляем данные методом POST, то, помимо всего прочего, должны указать метод их кодирования. Он задается как специфический параметр отправляемого запроса, носящий имя Content-type. А указать значение нужного параметра мы можем вызовом метода setRequestHeader объекта XMLHttpRequest:

```
<экземпляр объекта XMLHttpRequest>.setRequestHeader(<имя параметра>,
<значение параметра>)
```

Оба аргумента этого метода указываются в виде строк. Результата он не возвращает.

В нашем случае именем параметра станет строка "Content-type", а его значением — наименование нужного метода кодирования данных.

А собственно отправка запроса выполняется вызовом не возвращающего результата метода send объекта XMLHttpRequest.

Если отправка данных выполняется методом GET, метод send вызывается без аргументов. В этом случае отправляемые серверной программе данные указываются непосредственно в интернет-адресе, передаваемом методу open вторым аргументом.

Если же применяется метод POST, отправляемые данные указываются в вызове метода send его единственным аргументом. (Интернет-адрес, заданный в вызове метода open, не должен их содержать.) Эти данные указываются в том же формате, что и в случае метода GET, но без символа вопросительного знака в начале, и также в виде строки.

Для преобразования значений отдельных параметров к виду, пригодному для отправки, мы можем использовать встроенную в JavaScript функцию encodeURIComponent. В качестве единственного аргумента она принимает строку, которую следует соответственно закодировать, и возвращает ту же строку, но уже в закодированном виде.

В следующем примере мы отправляем Web-серверу синхронный запрос на получение файла /fragments/fragment3.html (понятно, что это фрагмент HTML-кода):

```
oAJAX.open("GET", "/fragments/fragment3.html", false);
oAJAX.send();
```

В листинге 22.1 приведен код, по нажатию кнопки **Найти** отправляющий серверной программе search.php асинхронным запросом значение, занесенное в поле ввода, в параметре с именем keyword посредством метода GET.

Листинг 22.1

```
<input type="text" id="txtKeyword">
<input type="button" id="btnSearch" value="Найти">
. . .
var txtKeyword = document.getElementById("txtKeyword");
```

```
var btnSearch = document.getElementById("btnSearch");
btnSearch.addEventListener("click", function() {
  oAJAX.open("GET", "search.php?keyword=" +
  encodeURIComponent(txtKeyword.value), true);
  oAJAX.send();
});
```

А листинг 22.2 представляет код, по нажатию кнопки **Войти** отправляющий серверной программе `login.php` асинхронным запросом занесенные в форму данные. При этом используются интернет-адрес серверной программы, методы отправки и кодирования данных, указанные в теге `<form>` этой формы.

Листинг 22.2

```
<form id="frmLogin" action="login.php" method="POST"
enctype="application/x-www-form-urlencoded">
  <p>Имя: <input type="text" name="login"></p>
  <p>Пароль: <input type="password" name="password"></p>
  <p><input type="submit" name="submit" value="Войти"></p>
</form>
. . .
var frmLogin = document.getElementById("frmLogin");
frmLogin.addEventListener("submit", function(evt) {
  var oForm = evt.target;
  var s = "";
  var cEls = oForm.elements;
  for (i in cEls) {
    if (s != "") s += "&";
    s += cEls[i].name + "=" + encodeURIComponent(cEls[i].value);
  }
  oAJAX.open(oForm.method, oForm.action, true);
  oAJAX.setRequestHeader("Content-type", oForm.enctype);
  oAJAX.send(s);
  evt.preventDefault();
});
```

Отметим, что в последнем примере мы фактически написали универсальный код, который можно использовать для отправки по технологии AJAX данных, введенных в любую форму. (Обратим также внимание, как свойство `target` события используется для получения элемента страницы, в котором возникло событие, в нашем случае — формы.) Но, разумеется, этот код придется несколько расширить, т. к. форма может содержать, помимо полей ввода, еще и флажки, переключатели и списки, которые нужно обрабатывать особо.

Описанным здесь способом мы можем отправить с применением AJAX любые данные, за исключением файлов. Для отправки файлов, указанных в форме, мы используем другой прием, который будет описан в *главе 43*.

Получение результата

Получив от нас запрос, выполненный по технологии AJAX, а также, возможно, какие-то данные, Web-сервер или серверная программа сформирует на его основе результат. Этим результатом будет либо запрошенный нами файл, либо данные, сгенерированные серверной программой. Как нам его получить? Из свойства `responseText` объекта `XMLHttpRequest`. Оно возвращает в виде строки полученный результат: фрагмент HTML-кода, данные, закодированные в форматах XML или JSON.

А вот момент, в который мы можем получить этот результат, зависит от того, синхронный или асинхронный запрос мы отправили ранее.

Если мы отправили *синхронный* запрос, то после обработки вызова метода `send` Web-обозреватель приостановит выполнение кода сценария до того момента, пока не будет получен запрошенный файл. А как только файл будет получен, он продолжит выполнение кода. Следовательно, мы можем обратиться к полученным данным сразу же после вызова метода `send`.

Пример обработки файла, полученного синхронным запросом, показан в листинге 22.3.

Листинг 22.3

```
<div id="output"></div>
. . .
oAJAX.open("GET", "/fragments/fragment3.html", false);
oAJAX.send();
var oOutput = document.getElementById("output");
oOutput.innerHTML = oAJAX.responseText;
```

Здесь мы получаем результат — запрошенный файл `/fragments/fragment3.html` с фрагментом HTML-кода — сразу же после вызова метода `send`, поскольку выполнили синхронный запрос.

В случае отправки *асинхронного* запроса Web-обозреватель продолжит выполнять код, следующий за вызовом метода `send`. А когда будет получен результат, в экземпляре объекта `XMLHttpRequest`, обрабатывающего запрос, возникнет событие `readystatechange`. В обработчике этого события мы и сможем получить доступ к полученным данным.

Здесь нужно сказать, что событие `readystatechange` возникает и при отправке синхронного запроса. Так что мы можем использовать один и тот же код для обработки запросов обоих типов.

Но есть небольшая проблема. Дело в том, что событие `readystatechange` возникает всякий раз, когда изменяется состояние ожидания результата (когда запрос собственно отправляется, когда результат получен, но еще не обработан, и др.). И еще нам следует иметь в виду, что Web-сервер или серверная программа может вернуть

не собственно запрашиваемый нами результат, а сообщение об ошибке, которая возникла при его обработке.

Так что нам в этом случае понадобятся следующие свойства объекта `XMLHttpRequest`:

- `readyState` — возвращает целочисленный признак состояния, в котором находится экземпляр объекта `XMLHttpRequest`:
    - 0 — запрос еще не выполнен;
    - 1 — запрос выполнен, но файл еще не получен;
    - 2 — файл получен, но еще не обработан;
    - 3 — идет обработка полученного файла;
    - 4 — файл обработан, и его содержимое может быть извлечено из свойства `responseText`;
- `status` — возвращает код ответа Web-сервера в виде числа: 200 (файл успешно получен), 403 (доступ к файлу запрещен) или 404 (запрошенный файл не найден).

Пример обработки файла, полученного асинхронным запросом, приведен в листинге 22.4.

Листинг 22.4

```
frmLogin.addEventListener("submit", function(evt) {
  . . .
  oAJAX.open(oForm.method, oForm.action, true);
  oAJAX.setRequestHeader("Content-type", oForm.enctype);
  oAJAX.addEventListener("readystatechange", function(evt) {
    if ((evt.target.readyState == 4) && (evt.target.status == 200)) {
      var s = evt.target.responseText;
      // Файл получен. Выполняем эту обработку и вывод
    }
  });
  oAJAX.send(s);
  evt.preventDefault();
});
```

Здесь мы переписали приведенный ранее пример с отправкой данных формы по технологии AJAX таким образом, чтобы полученный от серверной программы результат был благополучно обработан.

Формат JSON

Мы уже знаем, что можем получать по технологии AJAX данные в трех разных форматах: HTML, XML и JSON. Только какой формат в каких случаях использовать?

- Формат HTML прекрасно подходит для получения данных, уже подготовленных для вывода на страницу и не требующих дальнейшей обработки: фрагментов страниц, отдельных статей или их частей.
- Формат XML используется в случае, если нам требуется написать Web-интерфейс к уже существующему интернет-сервису, отправляющему данные именно в таком формате.
- Формат JSON, напротив, — идеальный выбор для случаев, если мы сами создаем и клиентскую, и серверную часть сайта.

Дело в том, что данные XML сами по себе имеют весьма большой объем, а для их обработки требуется довольно громоздкий и сложный код. Напротив, данные JSON компактны, а обработать их можно буквально одним выражением. Конечно же, Web-программисты не могут пройти мимо такой находки!

Формат *JSON* (JavaScript Object Notation, объектная нотация JavaScript) представляет собой запись в строковом виде инициализатора, объявляющего экземпляр объекта `Object` (см. *главу 16*). Есть лишь два исключения: во-первых, в данных JSON имена свойств также берутся в кавычки, а во-вторых, их значениями не могут быть функции (т. е. в экземплярах объекта, закодированных в JSON, могут присутствовать лишь свойства, но не методы).

Вот пример данных JSON:

```
{ "id": 11, "title": "Введение в AJAX" }
```

Они характеризуют отдельную статью, опубликованную на гипотетическом сайте. Статья имеет внутренний идентификатор `11` (свойство `id`) и заголовок `"Введение в AJAX"` (свойство `title`).

В листинге 22.5 представлен пример кодирования в формате JSON массива данных.

Листинг 22.5

```
{
  "status": 1,
  "data": [
    { "id": 11, "title": "Введение в AJAX" },
    { "id": 12, "title": "Программная реализация AJAX" },
    { "id": 13, "title": "Формат JSON" }
  ]
}
```

Здесь свойство `status` хранит числовой код состояния (1 обозначает отсутствие ошибок при генерировании данных серверной программой), а свойство `data` — сам массив данных, описывающих статьи.

В язык JavaScript встроен объект `JSON`. Статический метод `parse` этого объекта выполняет декодирование данных JSON:

```
JSON.parse(<данные JSON>)
```

`Данные JSON` передаются в виде строки (собственно, в виде строки мы и получим их из свойства `responseText` объекта `XMLHttpRequest`). А возвращает этот метод сгенерированный на их основе экземпляр объекта `Object`, который мы можем без проблем обработать в сценарии.

В листинге 22.6 приведен код, который загружает данные, взятые из листинга 22.5 и сохраненные в файле `data.json`, и выводит содержащийся в них список статей на страницу в виде набора абзацев. Каждый абзац содержит заголовок статьи и, в скобках, ее идентификатор.

Листинг 22.6

```
<div id="output"></div>
. . .
var oOutput = document.getElementById("output");
oAJAX = new XMLHttpRequest();
oAJAX.open("GET", "data.json", true);
oAJAX.addEventListener("readystatechange", function(evt) {
  if ((evt.target.readyState == 4) && (evt.target.status == 200)) {
    var oD = JSON.parse(evt.target.responseText);
    if (oD.status == 1) {
      var oP;
      for (i in oD.data) {
        oP = document.createElement("p");
        oP.textContent = oD.data[i].title + " (" + oD.data[i].id + ")";
        oOutput.appendChild(oP);
      }
    } else {
      oOutput.textContent = "Получены ошибочные данные.";
    }
  }
});
oAJAX.send();
```

AJAX-навигация

Технология AJAX открывает перед Web-программистами совершенно новые возможности в плане построения сайтов. И одновременно с этим позволяет значительно сократить объем данных, пересылаемых по сети (что может очень пригодиться пользователям мобильных устройств, выходящих в Интернет по каналам, где оплата начисляется за объем переданных данных).

Страницы, входящие в состав одного сайта, практически всегда включают в свой состав совершенно одинаковые фрагменты: «шапку» сайта, панель навигации и «поддон». Фактически эти страницы различаются лишь основным содержанием (которое, по определению, уникально для каждой страницы). И каждый раз, когда посетитель переходит на очередную страницу сайта, ему на компьютер раз за разом загружаются одни и те же фрагменты информации.

Можно ли загружать эти повторяющиеся фрагменты всего однажды — при посещении первой, главной, страницы сайта, а в дальнейшем получать из сети лишь основное содержание каждой страницы? Разумеется, можно, — применив технологию AJAX. Решение напрашивается само собой: при щелчке на гиперссылке вместо перехода на страницу должна выполняться подгрузка фрагмента HTML-кода с ее основным содержанием, который впоследствии помещается в нужный блок разметки.

В листинге 22.7 показан пример кода, реализующий такой подход.

Листинг 22.7

```
<nav>
  <p><a href="/design/html/basics.html">Основы HTML</a></p>
  <p><a href="/design/html/structuring.html">Структурирование
  текста</a></p>
  <p><a href="/design/html/formattind.html">Форматирование текста</a></p>
</nav>
<div id="main"></div>
. . .
var cNavLinks = document.querySelectorAll("nav a");
var oMain = document.getElementById("main");
var oAJAX = new XMLHttpRequest();
for (i in cNavLinks) {
  cNavLinks[i].addEventListener("click", function(evt) {
    evt.preventDefault();
    oAJAX.open("GET", evt.target.href, false);
    oAJAX.send();
    oMain.innerHTML = oAJAX.responseText;
  });
}
```

Но здесь нас подстерегает проблема. Интернет-адрес открытой в Web-обозревателе страницы при переходе на любую страницу такого сайта остается неизменным. Или, говоря другими словами, Web-обозреватель не «знает», что мы уже перешли на другую страницу, а не остаемся на главной.

Ничего страшного? Как бы не так... Если посетитель оставит закладку на какую-либо страницу такого сайта, а потом, спустя какое-то время, откроет ее, он окажется не на желаемой странице, а на главной. А если он решит воспользоваться стандартной кнопкой возврата на предыдущую страницу, то окажется на странице предыдущего сайта, с которой перешел на наш.

Но и эта проблема решаема. В *главе 5*, знакомясь со средствами навигации, мы узнали о существовании якорей, позволяющих перейти на определенное место страницы — неважно, текущей или какой-то другой.

Нас интересует случай, когда якорь находится на текущей странице. При щелчке на якоре Web-обозреватель формирует его полный интернет-адрес, состоящий из ин-

тернет-адреса самой страницы и имени якоря, которое добавляется к интернет-адресу страницы и отделяется от него символом решетки (#). После чего выполняется переход по полученному интернет-адресу.

При этом, с одной стороны, Web-обозреватель считает, что выполняется переход по другому интернет-адресу, заносит новый интернет-адрес в список истории и после нажатия кнопки возврата на предыдущую страницу осуществляет переход на позицию «назад» — к гиперссылке, указывающей на этот якорь. Но, с другой стороны, текущая страница не перезагружается. То, что нам нужно!

Исходя из сказанного, для реализации навигации по сайту в духе AJAX нам нужно лишь создать гиперссылки, в интернет-адресах которых задать какие-либо указатели на целевые страницы в формате имен якорей. Например, так, как показано в листинге 22.8.

Листинг 22.8

```
<nav>
  <p><a href="#/design/html/basics.html">Основы HTML</a></p>
  <p><a href="#/design/html/structuring.html">Структурирование
  текста</a></p>
  <p><a href="#/design/html/formattind.html">Форматирование
  текста</a></p>
</nav>
```

При щелчке на такой гиперссылке страница не будет перезагружена, но изменится часть текущего интернет-адреса, следующая за символом решетки. После чего в экземпляре объекта `Window`, представляющем текущее окно Web-обозревателя, возникнет событие `hashchange` (см. *главу 20*), которое мы можем обработать. Листинг 22.9 показывает код, выполняющий такую обработку.

Листинг 22.9

```
window.addEventListener("hashchange", function() {
  var href = window.location.hash;
  href = href.substr(1);
  oAJAX.open("GET", href, false);
  oAJAX.send();
  oMain.innerHTML = oAJAX.responseText;
});
```

Свойство `hash` объекта `Location` хранит имя якоря вместе с начальным символом решетки. От этого символа нам придется избавиться, ведь в противном случае будет выполнена попытка получения файла с именем, в начале которого присутствует символ решетки, и попытка эта, как нетрудно догадаться, не увенчается успехом. Мы используем уже знакомый нам по *главе 16* метод `substr` объекта `String`, извлекая с его помощью часть интернет-адреса, начиная со второго символа.

Как видим, код в этом случае получается заметно более компактным, и, к тому же, нам не придется привязывать обработчики событий ко всем гиперссылкам панели навигации. Так что мы имеем двойную выгоду.

В настоящее время определенную популярность получили сайты, состоящие из одной страницы, но имитирующие наличие нескольких страниц. Единственная страница такого сайта включает несколько блоков, каждый из которых представляет собой отдельную «страницу», и панель навигации, при щелчке на пункте которой особый сценарий выводит на экран соответствующий блок и скрывает все остальные. Код простейшего сайта подобного рода (их так и называют — *одностраничными*) приведен в листинге 22.10.

Листинг 22.10

```
#pages > div        { display: none; }
#pages > div.active { display: block; }
. . .
<p>
  <a href="#page1">Страница 1</a>,
  <a href="#page2">страница 2</a>,
  <a href="#page3">страница 3</a>
</p>
<div id="pages">
  <div id="page1" class="active">Содержимое страницы 1</div>
  <div id="page2">Содержимое страницы 2</div>
  <div id="page3">Содержимое страницы 3</div>
</div>
. . .
window.addEventListener("hashchange", function(evt) {
  var el = document.body.querySelector("#pages > div.active");
  el.className = "";
  el = document.getElementById(window.location.hash.substr(1));
  el.className = "active";
});
```

Все блоки, хранящие содержимое «страниц», мы поместили в блок `pages` и задали для них имена вида `page`*<номер "страницы">*, тем самым превратив их в якоря. Гиперссылки панели навигации указывают на эти якоря.

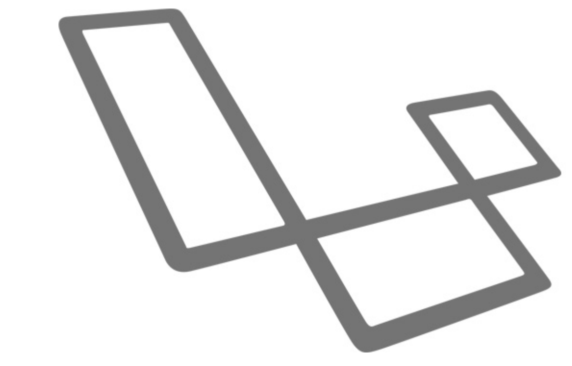
По умолчанию все блоки-«страницы» скрыты. Чтобы сделать какой-либо блок видимым, мы привяжем к нему стилевой класс `active`.

Обработчик события `hashchange` ищет блок-«страницу» с привязанным стилевым классом `active` и убирает его привязку, тем самым скрывая блок, бывший ранее видимым. После этого обработчик получает имя якоря, на который указывает активизированная посетителем гиперссылка, и удаляет из него начальный символ решетки. Сформировав таким образом имя блока, он ищет этот блок и привязывает к нему стилевой класс `active`, после чего блок становится видимым.

Осталось сказать, что способ навигации по сайту, описанный здесь, носит название *AJAX-навигации*.

ЧАСТЬ II

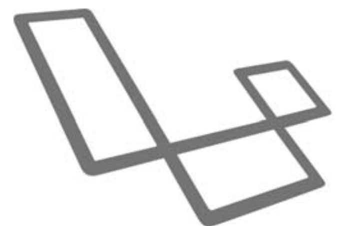
РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ WEB-САЙТА



II. РАЗДЕЛ 4

Введение в серверное программирование. PHP. MySQL

Глава 23.	Серверные программы. Фреймворки
Глава 24.	Программная платформа PHP
Глава 25.	Базы данных. Сервер данных MySQL



ГЛАВА 23

Серверные программы. Фреймворки

Все предыдущие разделы книги были посвящены разработке статических страниц и сайтов. Как мы помним из *главы 1*, статические страницы верстаются таким образом, чтобы они уже содержали всю информацию, ради которой они, собственно, создаются, сохраняются в файлах и более не изменяются (разумеется, пока их создатель не захочет что-либо переделать в них). Совокупность таких страниц составляет статический сайт.

Динамические страницы и сайты

Но в современном WWW господствует принципиально другой подход. Заключается он в следующем…

- Вся информация, которая публикуется в Сети посредством того или иного сайта, хранится в базе данных. Это, прежде всего, текстовая информация, а сопутствующие внедренные элементы: графические изображения, аудио- и видеоролики, архивы, дистрибутивы программ, входящие в состав этой информации, — хранятся, как и обычно, в отдельных файлах.
- Хранящаяся в базе данных информация преобразуется в обычные Web-страницы особыми программами, работающими совместно с Web-сервером, — *серверными программами*.
- Web-обозреватель отправляет Web-серверу запрос, содержащий интернет-адрес файла серверной программы, генерирующей требуемую страницу. Как правило, этот интернет-адрес включает в себя набор GET-параметров, однозначно указывающих, какую часть информации, хранящейся в базе, он желает получить в виде этой страницы.
- Web-сервер, получив запрос, извлекает из него путь к файлу серверной программы, запускает ее (вместо того, чтобы отправить этот файл Web-обозревателю) и передает ей полученные в составе интернет-адреса параметры.
- Серверная программа запрашивает из базы данных информацию, соответствующую значениям этих параметров, генерирует на ее основе страницу и посредством Web-сервера возвращает ее Web-обозревателю.

- Web-обозреватель даже не знает, что полученная им страница не считана из файла, а сгенерирована динамически особой программой. С его точки зрения — это обычная страница, ничем не отличающаяся от статической.

Как видим, страницы подобного рода сайтов динамически генерируются при каждом запросе. Они так и называются — *динамические Web-страницы*.

Совокупность динамических страниц, или, другими словами, серверных программ, формирующих их, представляет собой *динамический Web-сайт*.

Динамические сайты обладают перед статическими рядом существенных преимуществ:

- информация в базах данных хранится в «сыром», необработанном, виде и обрабатывается, а именно форматируется надлежащим образом, лишь при генерировании на ее основе Web-страницы. Что дает возможность отображать эту информацию на разных страницах различным образом.

  В качестве примера можно привести хранящиеся в базе данных сведения об опубликованных на сайте статьях — эти сведения включают в себя заголовки, анонсы, содержание статей и что-либо еще. Страница с перечнем статей выведет лишь заголовки и анонсы, к тому же, этот перечень может быть отфильтрован и отсортирован согласно различным критериям. А страница с самой статьей отобразит ее заголовок и содержание. Как видим, хранящиеся в базе «сырые» данные форматируются при выводе на экран различным образом — в зависимости от назначения страницы;
- чрезвычайно просто реализуется пополнение содержания сайта силами его пользователей.

  Применительно к статье пользователь просто заполняет Web-форму, вводя в нее заголовок, анонс, содержание статьи и, возможно, какие-то дополнительные данные (скажем, категорию, к которой относятся статья), и нажимает кнопку отправки данных. Особая серверная программа записывает новую статью в базу данных, после чего статья становится доступной для всех посетителей сайта.

  Таким же образом — посредством специально написанных серверных программ — можно без проблем реализовать выгрузку посетителями файлов на динамический сайт. Это могут быть файлы иллюстраций к статьям, дистрибутивов программ, мультимедийные файлы и проч.;
- опять же, чрезвычайно просто реализуется *разграничение доступа* — защита конфиденциальной информации (тех же «сырых», необработанных, текстов статей) от попадания в руки злоумышленников и просто лиц, не имеющих соответствующих полномочий.

  Разграничение доступа реализуется следующим образом:

  - создается список *зарегистрированных пользователей*, или просто *пользователей*, — посетителей, которые имеют право работать с внутренними данными сайта. Практически всегда такой список хранится в базе данных наряду с другой внутренними данными;

- посетитель, заходя на сайт, выполняет процедуру *входа*, в процессе которой вводит в особую форму имя, под которым он зарегистрирован в списке пользователей, и пароль. Серверная программа, реализующая вход, проверяет, есть ли в списке пользователь с таким именем и паролем, и выполняет вход только в том случае, если он там наличествует;
- пользователь может добавлять и править материалы сайта только после успешного выполнения процедуры входа;
- действия, которые пользователю разрешается выполнять на сайте, могут определяться его *правами*, которые также хранятся в списке пользователей. Так, можно позволить какому-либо пользователю только добавлять новые статьи, править и удалять статьи, написанные лишь им самим, а другому — править и удалять любые статьи;
- посетитель, не выполнивший вход на сайт, даже если он занесен в список пользователей, не может выполнять все эти действия. Такой посетитель называется *гостем*;
- завершив работу, пользователь выполняет процедуру *выхода* с сайта. После этого он теряет право работать с внутренними данными сайта до следующего входа. Выход с сайта обязательно следует выполнять, чтобы предотвратить доступ к внутренним данным сайта с компьютера этого пользователя;

❒ и, наконец, динамические сайты удобнее в сопровождении. Скажем, чтобы переделать дизайн страниц, достаточно внести исправления в код всего одной серверной программы, а не перелопачивать все страницы сайта.

Недостатков у динамических сайтов два. Во-первых, чтобы их создавать, недостаточно лишь знаний HTML, CSS и JavaScript — необходимы еще навыки в разработке программ, генерирующих страницы, и знания языков, на которых они пишутся. Во-вторых, если динамические сайты создаются с применением промежуточных платформ (наподобие PHP), для их публикации необходимо установить на серверный компьютер, помимо Web-сервера, еще и программное ядро соответствующей платформы. Но эти недостатки — ничто перед новыми открывающимися возможностями!

Разработка серверных программ. Фреймворки

Так как же разрабатываются серверные программы? Собственно, почти так же, как и программы любого другого типа, за тем исключением, что, поскольку они предназначены для работы совместно с Web-сервером, то должны удовлетворять определенным требованиям, изложенным в соответствующей документации.

Серверные программы реализуются в трех разновидностях:

❒ в виде обычных исполняемых файлов, например, exe- и dll-файлов Windows. Такие программы разрабатываются на компилируемых языках программирования: C++, Delphi, Visual Basic и др.

Единственное преимущество такого рода программ — высочайшая производительность (поскольку эти программы компилируются в двоичный код, непосредственно исполняемый процессором компьютера). Недостатками тут являются высокая сложность разработки, потребность в специальной программной среде (Microsoft Visual Studio, Embarcadero Delphi и т. п.) для их написания, привязка к платформе (так, исполняемые файлы Windows не могут работать в системе Linux и наоборот). В настоящее время серверные программы такого вида применяются редко и только в особых случаях;

- в виде обычных Web-страниц, содержащих вставки программного кода, которые, собственно, и реализуют логику серверных программ. Такие серверные программы носят название *серверных Web-страниц* и разрабатываются на программных платформах Microsoft ASP.NET, PHP и аналогичных.

  Их преимущества состоят в упрощении разработки (такие программы могут быть написаны с применением обычных текстовых редакторов — как и статические Web-страницы), в возможности объединения программного кода и HTML-кода страниц, в возможности создания серверных программ на основе уже готовых статических страниц. Существенный недостаток, пожалуй, всего один — меньшая производительность по сравнению с программами первого вида;

- на основе одного из существующих на рынке *фреймворков* — программных продуктов, предназначенных для разработки динамических сайтов и уже изначально реализующих значительную часть их функциональности. В этом случае серверные программы пишутся под ту же программную платформу, на которой был реализован выбранный для разработки фреймворк, и представляют собой отдельные программные модули, встраиваемые в этот фреймворк и выполняющие задачи, которые специфичны для конкретного сайта, а именно формирование конкретных страниц динамического сайта и некоторые вспомогательные действия.

  Преимущества у подобного рода программ — те же, что и у программ предыдущей разновидности. К ним добавляется радикальное упрощение разработки вследствие того, что значительная часть функциональности, которую нам потребуется реализовать в сайте, уже присутствует в составе самого фреймворка (к таким задачам относится взаимодействие с базой данных, разграничение доступа, защита от сетевых атак и проч.). Недостатком является некоторое снижение производительности, впрочем, не настолько значительное, чтобы принимать его в расчет.

В настоящее время подавляющее большинство динамических сайтов создается третьим способом — с применением фреймворков. Нам грядущий сайт будет разрабатываться таким же образом.

Введение во фреймворки. Модели, шаблоны, контроллеры

Если уж мы собрались разрабатывать сайт с помощью фреймворка, то следует сначала узнать, что это, вообще, такое.

По большему счету, фреймворк — набор (*библиотека*) серверных программ, реализующая более или менее значительную часть функциональности сайта. Обычно предпочтение отдается тем продуктам такого рода, которые реализуют как можно бóльшую часть необходимых функций.

Эти функции включают в себя:

- унифицированные инструменты для работы с базой данных, скрывающие особенности ее формата (о базах данных мы поговорим в *главе 25*);
- базовые средства для получения данных, отправленных посетителем, с целью использовать их для генерирования страниц;
- средства для упрощения формирования динамических страниц. Как правило, они позволяют использовать как основу полностью сверстанные статические страницы;
- инструменты для организации разграничения доступа (для ведения списка пользователей, выполнения процедуры входа, защиты страниц от доступа к ним посетителей, не выполнивших вход, и т. п.);
- инструменты для защиты от сетевых атак (для проверки данных, полученных от посетителя, на предмет вредоносных фрагментов и удаления таковых, для предотвращения доступа к внутренним данным сайта из других сайтов и др.);
- всевозможные вспомогательные средства (поддержка форматирования с применением тегов BBCode, защита данных с помощью CAPTCHA и т. п.);
- инструментальные средства для создания заготовок серверных программ, которые будут написаны разработчиком сайта, с целью упростить его работу (*прототипирования*).

Так что разработчику сайта останется лишь реализовать функциональность, специфичную для своего сайта: выборку данных из конкретной базы, обработку полученных данных по конкретным алгоритмам и формирование на основе полученных в результате обработки результатов конкретных динамических страниц. Для этого разработчик напишет соответствующие программные модули и встроит их во фреймворк.

Серверные программы, которые пишутся разработчиком, добавляются во фреймворк и реализуют специфическую функциональность сайта, делятся на несколько групп.

- *Модели* — используются для организации взаимодействия с отдельными таблицами базы данных. (О таблицах баз данных разговор будет вестись в *главе 25*.) На каждую таблицу создается одна модель.

  Модель служит своего рода посредником между данными, хранящимися в таблице, и другими серверными программами. Она предоставляет удобные средства как для выборки данных, так и для их правки, включая добавление и удаление.
- *Шаблоны* — образцы, на основе которых будут генерироваться динамические страницы. Представляют собой обычные страницы — и могут быть сделаны на

основе готовых статических страниц, — где в нужных местах кода вставлены особые инструкции, указывающие поместить сюда какие-либо данные.

Шаблоны определяют внешний вид страниц сайта. Более никаких задач они не выполняют.

- *Контроллеры* — запускаются при получении запроса с интернет-адресом определенного вида, получают отправленные посетителем данные, извлекают другие данные из базы посредством моделей, обрабатывают их и генерируют результирующие страницы с применением указанного шаблона.

  Контроллер представляет собой класс, поддерживающий набор методов. Каждый из таких методов либо выводит на экран страницу, либо обрабатывает данные, присланные посетителем или извлеченные из базы. Эти методы носят название *действий*.

  Как правило, один контроллер реализует функциональность одного раздела сайта, а его действия — различные части этой функциональности.

  Контроллеры и их действия выполняют активную работу по обработке данных и формированию страниц сайта. Все прочие программы играют подчиненную роль.

- *Маршрутизатор* — выполняет разбор интернет-адреса, полученного в результате запроса, и определяет, какое действие какого контроллера следует выполнить и, соответственно, какую страницу следует сгенерировать.

  Маршрутизатор в составе сайта всего один. И обычно уже поставляется в составе фреймворка — разработчику сайта остается лишь настроить его соответствующим образом.

- *Миграции* — выполняют создание и изменений структур базы данных, в которых хранится внутренняя информация сайта. (Впрочем, необходимые действия можно выполнить и с помощью соответствующих инструментальных средств, наподобие описанной в *приложении 2* программе phpMyAdmin.)

- *Файлы конфигурации* — хранят настройки сайта (параметры для подключения к базе данных, пути к ключевым папкам, особенности функционирования различных подсистем фреймворка и др.). Таких файлов может быть несколько.

- *Вспомогательные модули.* Они могут выполнять проверку введенных посетителем данных на корректность, указывать правила, согласно которым реализуется разграничение доступа, выполнять какие-либо действия при наступлении определенных условий (аналоги обработчиков событий в JavaScript) и др.

Большинство фреймворков предоставляют также и инструментальные средства для генерирования своего рода заготовок программ различных типов (прототипирования). Разработчик задает начальные параметры создаваемой программы, запускает процесс генерирования и получает заготовку, в которую ему остается лишь занести необходимый код.

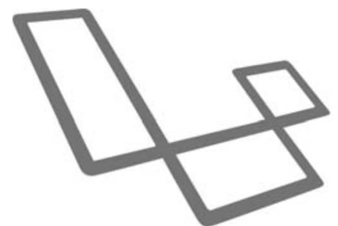
Еще нужно отметить, что некоторые возможности фреймворков реализованы в виде отдельных модулей, которые требуется устанавливать отдельно. Обычно в таком виде реализованы инструменты, применяемые только в особых случаях.

В настоящее время на рынке присутствует большое количество разнообразных фреймворков. Они написаны с применением самых разных программных платформ и, тем не менее, имеют схожее назначение и примерно одинаковый набор возможностей. Давайте вспомним хотя бы некоторые из них:

- Laravel, CodeIgniter, Symfony, Zend, Yii — написаны с применением платформы PHP;
- Django, Tornado, Pyramid — написаны на языке Python;
- Rails, Sinatra — написаны на языке Ruby;
- AngularJS, VueJS, ReactJS — написаны на языке JavaScript. Это весьма специфический вид фреймворков, исполняемых не на стороне сервера, а на стороне клиента.

Для разработки нашего сайта мы воспользуемся фреймворком Laravel, написанным с применением программной платформы PHP. Это самый популярный на данный момент PHP-фреймворк, предоставляющий огромное количество инструментальных средств на все случаи жизни и по-настоящему упрощающий разработку сайтов.

Мы начнем разговор о Laravel и разработку сайтов на его основе в *главе 26*. Предварительно нам необходимо познакомиться с самой платформой PHP, на которой написан этот фреймворк и на которой будет вестись разработка сайта, а также с сервером данных MySQL, который мы задействуем для хранения внутренних данных нашего сайта. PHP будет посвящена *глава 24*, а MySQL — *глава 25*.



ГЛАВА 24

Программная платформа PHP

Раз уж мы выбрали в качестве основы будущего сайта фреймворк Laravel, то в обязательном порядке должны познакомиться с программной платформой PHP. Сам Laravel написан на этой платформе, и, следовательно, серверные программы для сайта также будут писаться под нее.

PHP (Personal Home Page, личная домашняя страница) — популярная программная платформа, предназначенная для разработки серверных программ. С одной стороны, она предоставляет многочисленные программные инструменты, могущие покрыть практически все потребности Web-разработчика, а с другой, позволяет писать программы быстро, без особого труда и без необходимости в каких-либо специфических инструментальных средствах — достаточно лишь обычного текстового редактора.

Программы под PHP пишутся на языке, который так и называется — PHP. Сохраняются они в обычных текстовых файлах и могут включать как один лишь PHP-код, так и HTML-код со вставками PHP-кода.

В этой главе мы будем говорить о языке PHP, его возможностях и основных принципах написания на нем серверных программ.

В целом, язык PHP очень похож на язык JavaScript, который мы уже знаем. Настолько похож, что эта глава, по большей части, будет посвящена отличиям PHP от JavaScript.

Основные принципы, типы данных, переменные и операторы

- Простые типы данных: строки, числа, логические величины — в PHP не являются объектами. Следовательно, мы не можем обращаться к их свойствам и вызывать у них методы.

  Для работы с простыми типами данных PHP предоставляет большое количество встроенных функций. Их описание можно найти в справочнике на «домашнем» сайте PHP: **http://www.php.net/**.

- Имена переменных в PHP всегда предваряются символом доллара (`$`). Так, `$id` — правильное имя переменной, а `id` — неправильное (скорее всего, PHP посчитает ее функцией или классом).
- Если JavaScript является полностью регистрозависимым языком, то в PHP регистр символов имеет значение лишь при написании имен переменных. Так, `$id` и `$ID` — разные переменные, но `echo` и `ECHO` выполняют вызов одного и того же оператора `echo`.
- В PHP не используется оператор объявления переменных `var`. Переменные объявляются простым присвоением им значения, например:

  ```
  $id = 0;
  ```

- PHP допускает создание однострочных комментариев, предваряемых символом решетки (`#`):

  ```
  # Это комментарий
  ```

- Оператор объединения строк в PHP обозначается не символом `+`, а точкой (`.`):

  ```
  $s1 = "PHP";
  $s2 = "MySQL";
  $s = $s1 . " + " . $s2;
  ```

  В переменной `$s` окажется строка `"PHP + MySQL"`.

  Также можно использовать оператор сложного присваивания `.=`.
- *Оператор вывода* `echo` выводит на экран значения, указанные за ним и перечисленные через запятую. Вот несколько примеров:
  - выводим на экран строку `"PHP"`:

    ```
    echo "PHP";
    ```

  - выводим на экран значение переменной `$n` — число 8:

    ```
    $n = 8;
    echo $n;
    ```

  - выводим на экран строку `"PHP + MySQL"`:

    ```
    echo "PHP", " + ", "MySQL"
    ```

  Еще поддерживается оператор вывода — `print`, вызываемый точно так же, как `echo`, но он обрабатывается несколько медленнее и позволяет выводить лишь строки.
- PHP позволяет объявлять *именованные константы* — константы, к которым можно обращаться по имени. Для этого применяется встроенная функция `define`:

  ```
  define(<имя объявляемой константы>, <значение объявляемой константы>
  [, <имя константы зависит от регистра?>)
  ```

  Имя объявляемой константы указывается в виде строки (обычно имена констант набираются прописными буквами). Если третьим аргументом передано значение

false или если третий аргумент вообще не указан, при обращении к объявленной константе по имени не будет учитываться регистр символов, если true — будет.

В следующем примере мы объявляем именованную константу PLATFORM со значением "PHP":

```
define("PLATFORM", "PHP", true);
```

Получить доступ к объявленной именованной константе мы можем, указав ее имя без символа доллара.

В следующем примере мы выводим значение объявленной ранее именованной константы PLATFORM на экран:

```
echo PLATFORM;
```

Управляющие конструкции

Цикл по свойствам объекта JavaScript в PHP работает только с массивами, отчего и носит название *цикла по массиву*. Он записывается в следующем формате:

```
foreach (<массив> as <переменная>) {
  <тело цикла>
}
```

При этом в указанную *переменную* будет помещаться значение очередного элемента массива, а не его индекс, как в JavaScript.

Если нам нужно получить и индекс, и значение элемента массива (это может пригодиться при обработке ассоциативных массивов, о которых будет рассказано далее), мы используем расширенный формат цикла по массиву:

```
foreach (<массив> as <индекс> => <переменная>) {
  <тело цикла>
}
```

Вот пара примеров:

- после выполнения этого кода в переменной $s окажется сумма значений всех элементов массива $arr:

```
$s = 0;
foreach ($arr as $x) {
  $s += $x;
}
```

- другой вариант приведенного примера, в котором для доступа к текущему элементу массива используется его индекс:

```
$s = 0;
foreach ($arr as $i => $x) {
  $s += $arr[$i];
}
```

Функции

- В PHP в выражении объявления функции можно указать типы принимаемых ей параметров. Для этого достаточно поставить обозначение типа перед именем параметра. Пример:

  ```
  function someFunc(string $param) { . . . }
  ```

  Эта функция принимает параметр строкового типа. При попытке передать ей при вызове аргумент типа, отличного от строкового, возникнет ошибка.

- Необязательные параметры в функциях создаются с применением синтаксиса, аналогичного объявлению переменной — параметру просто присваивается его значение по умолчанию. Пример:

  ```
  function somefunction($a, $b = 10) { . . . }
  ```

  После чего, если при вызове функции `somefunction` не указать второй аргумент (`$b`), он получит значение по умолчанию 10.

  > **ВНИМАНИЕ!**
  >
  > Еще знакомясь с языком JavaScript в *главе 16*, мы узнали, что необязательными можно сделать только последние параметры функции. Однако, изучая фреймворк Laravel в последующих главах, мы столкнемся с методами, в которых необязательными являются не последние параметры, а расположенные в середине списка. Здесь нет никакого противоречия с предыдущим утверждением: необязательными в таких методах действительно являются последние параметры, просто при реализации метода используются некоторые программные трюки.

- В PHP любые глобальные переменные изначально недоступны внутри тела любой функции. При попытке внутри тела функции выполнить обращение к глобальной переменной будет создана одноименная ей локальная переменная, к которой и произойдет обращение.

  Перед тем, как в теле функции обратиться к глобальной переменной, последнюю следует явно объявить как глобальную. Для этого применяется *оператор объявления глобальных переменных* `global`, за которым через запятую перечисляются имена глобальных переменных.

  В листинге 24.1 показан пример объявления функции, в теле которой используется глобальная переменная `$globalvar`.

  Листинг 24.1

  ```
  $globalvar = 100;
  function somefunction($par) {
    global $globalvar;
    $globalvar = $par + 10;
  }
  ```

- И в JavaScript, и в PHP локальные переменные удаляются из памяти по завершении выполнения тела функции. Однако PHP позволяет нам создавать так назы-

ваемые *статические переменные*, которые сохраняются после выполнения тела функции и могут быть использованы при его последующих выполнениях. Такие переменные объявляются с помощью *оператора объявления статической переменной* `static`, за которым через запятую перечисляются объявления статических переменных.

Листинг 24.2 показывает объявление функции, в теле которой объявлена статическая переменная `$staticvar`. Три последовательных вызова этой функции будут выводить на экран числа 1, 2 и 3.

Листинг 24.2

```
function testfunction() {
  static $staticvar = 0;
  $staticvar++;
  echo $staticvar;
}
testfunction();
testfunction();
testfunction();
```

Массивы. Ассоциативные массивы

Массивы в PHP создаются двумя способами:

- в стиле PHP — с применением встроенной функции `array`. В качестве ее аргументов указываются значения элементов будущего массива, таких аргументов может быть произвольное количество. А возвращает эта функция готовый массив. Примеры:

  ```
  $arr = array(1, 2, 45, 784);
  $strings = array("HTML", "CSS", "JavaScript", "PHP", "MySQL");
  $emptyarray = array();
  ```

  Последнее выражение создает пустой массив;
- в стиле JavaScript — с применением квадратных скобок:

  ```
  $arr = [1, 2, 45, 784];
  $strings = ["HTML", "CSS", "JavaScript", "PHP", "MySQL"];
  $emptyarray = [];
  ```

Встроенная функция `count` возвращает количество элементов в массиве, переданном ей единственным аргументом. Пример:

```
if (count($strings) > 0) {
  // Массив не пуст
} else {
  // Массив пуст
}
```

Еще PHP позволяет создать массив, в качестве индексов элементов которого используются не целые числа, как обычно, а строки. Такой массив носит название *ассоциативного*, или *хэша*, а индексы его элементов называются *ключами*.

Для создания ассоциативного массива применяется та же функция `array`, однако в качестве ее аргументов указываются пары вида:

```
<ключ элемента> => <значение элемента>
```

Пример:

```
$software = array("platform" => "PHP", "database" => "MySQL");
```

В переменной `$software` окажется массив с двумя элементами, первый из которых будет иметь ключ "`platform`" и значение "`PHP`", а второй — ключ "`database`" и значение "`MySQL`".

Создать ассоциативный массив можно и с применением квадратных скобок:

```
$software = ["platform" => "PHP", "database" => "MySQL"];
```

Обратиться к элементу такого массива можно, указав в квадратных скобках его ключ:

```
$myPlatform = $software["platform"];
```

Здесь мы извлекаем и помещаем в переменную `$myPlatform` элемент массива `$software` с ключом "`platform`" — строку "`PHP`".

PHP даже позволяет совместить в одном массиве элементы с числовыми индексами и элементы с ключами, создав тем самым *комбинированный массив*. Пример:

```
$framework = ["Laravel", "platform" => "PHP", "dataFormat" => "MySQL"];
```

Здесь первый элемент массива (строка "`Laravel`") доступна по числовому индексу:

```
$fr = $framework[0];
```

а остальные — по их ключам:

```
$pl = $framework["platform"];
$dbf = $framework["dataFormat"];
```

В PHP-программировании подобного рода массивы применяются довольно часто.

Регулярные выражения

- Регулярные выражения в PHP указываются в виде строк. Они заключаются в любые парные символы, кроме обратного слеша: прямые слеши, «решетки», тильды и проч. Модификаторы, как и в случае JavaScript, ставятся после последнего такого символа.

  Примеры:

  ```
  $reg_exp1 = "/PHP\d+/i";
  $reg_exp2 = "#html?#";
  $reg_exp3 = "~Глава (\d+)~";
  ```

Для работы с регулярными выражениями PHP предоставляет целый набор встроенных функций.

- PHP поддерживает расширенный набор литералов и модификаторов регулярных выражений. Например, модификатор `s` указывает, что литерал `.` (точка) должен совпадать с любым символом, включая возврат каретки и перевод строки (по умолчанию этот литерал не совпадает с этими символами).

Классы и объекты

PHP, как и JavaScript, поддерживает объекты. Вот только терминология здесь используется совсем другая:

- то, что в JavaScript называется экземпляром объекта, в PHP носит название *объекта*,
- а то, что в JavaScript называется объектом, в PHP носит название *класса*.

> ***На заметку***
>
> Нужно отметить, что эта терминология применяется во всех объектно-ориентированных языках программирования. JavaScript — вероятно, единственное исключение из этого правила.

В целом, классы и объекты PHP имеют те же возможности и обрабатываются так же, как аналогичные им сущности JavaScript. Однако здесь имеется довольно много отличий, которые мы сейчас рассмотрим.

Первое отличие заключается в том, что, если при создании объекта не требуется указывать никаких аргументов, круглые скобки, что ставятся после имени класса, можно опустить. Пример:

```
$oObj1 = new SomeClass();
$oObj2 = new SomeClass;
```

Доступ к свойствам и методам объекта

В JavaScript, чтобы обратиться к свойству или методу экземпляра объекта, мы записывали имя переменной, где хранится этот экземпляр объекта, ставили точку, после чего указывали имя нужного свойства или метода. В PHP вместо точки применяется комбинация символов `->`. А имя свойства при этом указывается без символа доллара.

Примеры:

```
$oObj = new SomeClass();
$oObj->someProperty = 3;
$oObj->someMethod($a, $b, $c);
```

Для доступа к статическому свойству или методу мы напишем имя класса, которому они принадлежат, поставим символы `::` (два двоеточия) и запишем имя нужного свойства или метода, причем имя свойства предварим символом `$`:

```
SomeClass::$staticProperty = 10;
SomeClass::staticMethod();
```

Эта особенность PHP часто служит причиной ошибок, допускаемых даже опытными программистами. Поэтому всегда нужно помнить о ней.

Объявление классов

В следующих главах, изучая фреймворк Laravel, мы столкнемся с тем, что практически все входящие в его состав серверные программы представляют собой классы. Также в виде классов реализуются программные модули, содержащие специфичный для создаваемого сайта код. Поэтому нам обязательно нужно научиться объявлять новые классы, иначе мы недалеко продвинемся на ниве серверного программирования.

Простейший формат объявления класса таков:

```
class <имя класса> {
  <объявления свойств>
  <объявления методов>
}
```

Имя класса должно быть уникальным. Свойства и методы объявляются так же, как обычные переменные и функции.

Переменная `$this`, доступная в теле методов класса, хранит ссылку на текущий объект этого класса. Мы можем использовать ее, чтобы получить доступ к свойствам и методам этого объекта.

Листинг 24.3 представляет код, объявляющий класс `SomeClass` со свойством `someProperty` и методом `someMethod`.

Листинг 24.3

```
class SomeClass {
  $someProperty = 0;
  function someMethod($a, $b, $c) {
    $this->someProperty += ($a + $b) / $c;
  }
}
```

Теперь мы можем создать объект только что объявленного класса и использовать его в вычислениях:

```
$oSC = new SomeClass();
$oSC->someProperty = 1;
$oSC->someMethod(3, 5, 4);
```

Если в классе требуется объявить статическое свойство или метод, следует перед его именем вставить ключевое слово `static` и пробел.

Листинг 24.4 представляет объявление класса `SomeClass2` с обычным свойством `property`, обычным методом `method`, статическим свойством `staticProperty` и статическим методом `staticMethod`.

Листинг 24.4

```
class SomeClass2 {
  $property = 0;
  static $staticProperty = 7000;
  function method() { . . . }
  static function staticMethod() { . . . }
  . . .
}
```

Если же нам потребуется получить доступ к статическому свойству или методу изнутри того же класса, где они объявлены, мы используем переменную `self` вместо `$this`, поставим после этой переменной символы `::` (два двоеточия) и имя нужного свойства или метода. Отметим также, что имя переменной `self` пишется без символа `$`.

Листинг 24.5 показывает часть объявления класса `SomeClass2`, в котором в теле метода выполняется доступ к статическому свойству.

Листинг 24.5

```
class SomeClass2 {
  static $staticProperty;
  . . .
  function method() {
    . . .
    echo self::$staticProperty;
    . . .
  }
}
```

Наследование классов

Наследование — это порождение одного класса от другого. Новый класс (его называют классом-*потомком*) получает поддержку всех свойств и методов того класса, на основе которого он создан (класса-*родителя*).

Объявить класс-потомок другого класса можно, записав его объявление в формате:

```
class <имя класса-потомка> extends <имя класса-родителя> {
  <объявления свойств>
  <объявления методов>
}
```

Код из листинга 24.6 объявляет класс OtherClass — потомок объявленного ранее класса SomeClass. В новом классе имеется свойство otherProperty и метод otherMethod, и, вместе с тем, наличествует поддержка свойства someProperty и метода someMethod, унаследованных от класса SomeClass.

Листинг 24.6

```
class OtherClass extends SomeClass {
  $otherProperty = 1000;
  function otherMethod($a, $b, $c) {
    $this->otherProperty -= ($a + $b) / $c;
  }
}
```

Мы можем объявить в классе-потомке метод с тем же именем, что и у метода класса-родителя, в результате чего метод класса-потомка полностью заменит одноименный метод класса-родителя.

Однако чаще возникает потребность не изменить функциональность метода класса-потомка по сравнению с одноименным методом класса-родителя полностью, а слегка дополнить ее. Или, как говорят программисты, *переопределить* метод. Для этого в методе класса-потомка мы напишем код, реализующий дополненную функциональность, и в нужном его месте (перед написанным нами кодом или после него) вызовем этот же метод класса-родителя. Для этого мы применим формат вида:

parent::<*вызов метода*>

Отметим, что вызов метода записывается здесь без указания переменной $this и символов ->.

Листинг 24.7 показывает пример объявления класса ThirdClass — потомка класса OtherClass с переопределенным методом otherMethod. В последнем добавлена проверка, содержит ли свойство otherProperty ненулевое значение, и, если это не так, свойству присваивается его изначальное значение (1000). И лишь после выполнения такой проверки вызывается метод otherMethod класса-родителя.

Листинг 24.7

```
class ThirdClass extends OtherClass {
  function otherMethod($a, $b, $c) {
    if ($this->otherProperty <= 0) {
      $this->otherProperty = 1000;
    }
    parent::otherMethod($a, $b, $c);
  }
}
```

Наследование классов — исключительно мощный инструмент, позволяющий разделить функциональность между классами. Общие функции мы можем поместить

в классы-родители, от которых породить классы-потомки, выполняющие более специфические функции. Все фреймворки, предназначенные для разработки динамических сайтов, в том числе и Laravel, активно используют наследование.

Конструкторы и деструкторы

Вспомним, как мы создавали экземпляры объектов на языке JavaScript в *разделе 3*. Мы записывали оператор `new`, после него ставили имя объекта, а затем в круглых скобках перечисляли значения необходимых аргументов (если объект требовал аргументы). Похоже на вызов функции, не так ли?

Но на самом деле это и есть вызов специфической функции, точнее, метода объекта, называемого *конструктором* класса. Он вызывается всякий раз, когда создается экземпляр какого-либо объекта.

В PHP создание объектов на основе класса происходит аналогичным образом — вызовом его конструктора. Конструктор класса имеет имя `__construct`, может принимать произвольное количество параметров и не должен возвращать результат.

Листинг 24.8 показывает часть объявления класса `ThirdClass`, включающую объявление его конструктора. Конструктор принимает два параметра и сохраняет их значения в свойствах `someProperty` и `otherProperty`.

Листинг 24.8

```
class ThirdClass extends OtherClass {
  function __construct($some, $other) {
    $this->someProperty = $some;
    $this->otherProperty = $other;
  }
  . . .
}
```

Теперь мы можем создать объект класса `ThirdClass` с применением нового конструктора:

```
$oTC = new ThirdClass(20, 400);
```

Конструкторы также наследуются от классов-родителей и их также можно переопределять, как и другие методы.

Если мы не создадим конструктор в классе, последний получит простейший конструктор по умолчанию, который не принимает аргументов и не выполняет никаких действий. Он-то и будет вызван, когда мы выполним создание класса без указания аргументов:

```
$oSC = new SomeClass;
```

Иногда бывает необходимо выполнить какие-либо действия перед тем, как объект будет уничтожен. Эти действия мы можем реализовать в методе, называемом

деструктором класса. Деструктор выполняется непосредственно перед уничтожением объекта, имеет имя `__destruct`, не принимает параметров и так же, как и конструктор, не возвращает результата.

Пример:

```
class ThirdClass extends OtherClass {
  . . .
  function __destruct() {
    . . .
  }
}
```

Вообще-то, в классе `ThirdClass` деструктор не очень-то и нужен, но это лишь пример.

Модификаторы доступа

Модификатор доступа — это особый признак, указывающий, откуда будет доступно свойство или метод класса. Модификатор доступа ставится перед именем свойства и метода, перед ключевым словом `static`, если оно присутствует, и отделяется от них пробелом.

PHP поддерживает три модификатора доступа:

- `public` — свойство (метод) доступно везде: вне класса, внутри текущего класса, во всех его классах-потомках и в других классах (*публичное* свойство или метод);
- `protected` — свойство (метод) доступно только внутри текущего класса и во всех его классах-потомках (*защищенное* свойство или метод);
- `private` — свойство (метод) доступно только внутри текущего класса (*частное* свойство или метод).

Если модификатор доступа не указан, к свойству или методу можно получить доступ отовсюду, как если бы был указан модификатор `public`.

Пример использования модификаторов доступа представлен в листинге 24.9.

Листинг 24.9

```
class SomeClass {
  $property1;
  public $property2;
  protected function method1() { . . . };
  private function method2() { . . . };
}
```

Свойства `property1` и `property2` будут доступны везде, метод `method1` — в текущем классе и классах-потомках, а метод `method2` — только в текущем классе.

Модификаторы доступа помогут нам скрыть часть функциональности класса, оставив ее, так сказать, для внутреннего пользования. Это может быть полезно, если мы разрабатываем какой-либо код, предназначенный для использования сторонними программистами.

Константы класса

В начале этой главы мы узнали, что PHP позволяет объявлять именованные константы. Такую константу мы можем сделать составной частью класса, создав *константу класса*.

Константа класса создается так же, как обычное свойство, за двумя исключениями: объявление константы должно предваряться словом `const`, а ее имя записывается без символа доллара.

Пример:

```
class OtherClass {
  const PLATFORM = "PHP";
  . . .
}
```

Доступ к константам класса выполняется так же, как и к статическим свойствам, только имя константы записывается без символа доллара:

```
echo OtherClass::PLATFORM;
```

Интерфейсы

Интерфейсом в PHP называется сущность, объединяющая в себе набор объявлений методов без их реализации (то есть без тела). Интерфейс выступает своего рода указанием классу, какие методы он должен реализовать.

Интерфейсы объявляются так же, как классы, за следующими исключениями:

- ❒ вместо ключевого слова `class` указывается ключевое слово `interface`;
- ❒ интерфейс должен включать лишь объявления методов без их тела, при этом не указываются даже фигурные скобки, в которых обычно заключается код тела метода;
- ❒ все методы, входящие в состав интерфейса, должны быть публичными;
- ❒ интерфейсы не могут включать свойства;
- ❒ тем не менее, интерфейсы могут включать константы.

Листинг 24.10 показывает пример объявления интерфейса `SomeInterface`, включающего один метод `someMethod`.

Листинг 24.10

```
interface SomeInterface {
  public function someMethod(a, b);
}
```

Интерфейсы, как и классы, могут наследоваться друг другом. Листинг 24.11 показывает пример объявления интерфейса `OtherInterface` — потомка `SomeInterface`, который включает константу `SOMECONST`, метод `otherMethod` и наследует от родителя метод `someMethod`.

Листинг 24.11

```
interface OtherInterface extends SomeInterface {
  constant SOMECONST = "SOMECONST";
  public function otherMethod(c, d, e);
}
```

Объявление класса, реализующего интерфейс, записывается в следующем формате:

```
class <имя класса-потомка> [extends <имя класса-родителя>]
implements <список реализуемых интерфейсов, разделенных запятыми> {
  <объявления свойств>
  <объявления методов класса>
  <реализация методов интерфейсов>
}
```

Отметим, что реализуемые в классе методы должны быть публичными, иметь то же имя и тот же набор параметров, что и одноименные методы интерфейса. Отметим также, что класс может реализовывать произвольное количество интерфейсов.

Листинг 24.12 показывает объявление двух классов: первый класс реализует интерфейс `SomeInterface`, а второй — интерфейсы `SomeInterface` и `OtherInterface`.

Листинг 24.12

```
class SomeClass3 implements SomeInterface {
  . . .
  public function someMethod(a, b) { . . . }
}
class SomeClass4 implements SomeInterface, OtherInterface {
  . . .
  public function someMethod(a, b) { . . . }
  public function otherMethod(c, d, e) { . . . }
}
```

Интерфейсы обычно применяются для определения функциональности (набора методов и констант), которая должна быть общей у разных классов, необязательно являющимися потомками одного предка.

Трейты

Трейт можно рассматривать как более развитую разновидность интерфейса. В отличие от последнего, он включает методы вместе с реализацией (телом), в трейтах также допускается объявлять свойства.

Трейты объявляются так же, как классы, только вместо ключевого слова `class` ставится ключевое слово `trait`. Наследоваться от других классов или трейтов, равно как и реализовывать интерфейсы, трейты не могут.

Листинг 24.13 представляет код объявления трейта `SomeTrait`, содержащего метод `someMethod`.

Листинг 24.13

```
trait SomeTrait {
  public function someMethod(a, b) { . . . }
}
```

После этого трейт можно добавить в любой класс, и этот класс станет поддерживать все свойства и методы, объявленные в трейте. Добавить трейт к классу можно, вставив в любом месте объявления класса выражение вида:

```
use <список добавляемых к классу трейтов, разделенных запятыми>
```

Пример:

```
class SomeClass {
  use SomeTrait;
  . . .
}
```

Если в добавленном в класс трейте и непосредственно в этом же классе присутствуют методы с одинаковыми именами, то при обращении по этому имени будет вызван метод, объявленный в классе. Если же имя метода трейта совпадает с именем метода, унаследованного от класса-родителя, при обращении по этому имени будет вызван уже метод трейта.

Пространства имен

Пространство имен в PHP — сущность, объединяющая произвольное количество функций, констант, классов, интерфейсов и трейтов. Разные пространства имен могут включать в себя функции, константы, классы, интерфейсы и трейты с одинаковыми именами.

Пространства имен активно используются в Laravel для объединения классов, выполняющих сходные задачи.

Определение пространств имен

Физически пространство имен представляет собой папку, в которой хранятся файлы с программным кодом. Следовательно, для того чтобы создать пространство имен, нам потребуется:

- создать папку с именем, которое должно совпадать с именем этого пространства имен;

- переместить в нее все файлы, что должны входить в это пространство имен;
- записать в самом начале каждого из этих файлов выражение определения пространства имен вида:

```
namespace <имя пространства имен>
```

Пример объявления пространства имен в начале программного файла:

```
// Файл хранится в папке NameSpace1
namespace NameSpace1;
. . .
```

Пространства имен могут быть вложены друг в друга. Чтобы создать вложенное пространство имен, в папке уже существующего пространства имен следует создать папку, которая определит вложенное пространство имен, и записать в начале каждого входящего в него файла приведенное ранее выражение. В этом выражении в качестве имени пространства имен должен быть указан своего рода путь к нему, в котором отдельные имена пространств имен разделяются символами обратного слеша.

Пример объявления вложенного пространства имен в начале программного файла:

```
// Файл хранится в папке NameSpace1\NameSpace2
namespace NameSpace1\NameSpace2;
. . .
```

Уровень вложенности пространств имен друг в друга не ограничен. В современных фреймворках может быть большое количество пространств имен, многократно вложенных друг в друга.

Работа с пространствами имен

Если в каком-либо программном файле требуется вызвать функцию или константу, создать объект класса, использовать интерфейс или трейт, объявленный в другом пространстве имен, следует записать *полное имя* одной из указанных сущностей. Полное имя представляет собой перечисление наименований пространств имен, в которые последовательно вложен файл с объявлением, в направлении от «внешних» к «внутренним», и имя самой вызываемой сущности (функции, константы, класса, интерфейса или трейта), разделенные обратными слешами.

Предположим, у нас есть файл в пространстве имен `NameSpace1\NameSpace2`, который объявляет функцию `someFunc`:

```
namespace NameSpace1\NameSpace2;
. . .
function someFunc() { . . . }
```

И есть файл, хранящийся в папке, в которую вложено самое «внешнее» из пространств имен — `NameSpace1`. В этом файле требуется вызвать функцию `someFunc`. Для этого мы запишем ее вызов следующим образом:

```
$result = NameSpace1\NameSpace2\someFunc();
```

Здесь мы последовательно перечислили имена пространства имен, в которые вложено объявление этой функции, и указали имя самой функции, разделив их обратными слешами.

Если требуется обратиться к функции `someFunc` из файла, что находится в пространстве имен `NameSpace1`, ее вызов будет записан так:

```
namespace NameSpace1;
$result = NameSpace2\someFunc();
```

Если нет желания каждый раз писать длинное полное имя, можно выполнить операцию *импортирования*. Импортировать можно класс, интерфейс, трейт и пространство имен. Функцию или константу импортировать нельзя.

Для выполнения импортирования применяется выражение вида:

```
use <полное имя импортируемой сущности> [as <псевдоним>]
```

После его выполнения можно будет получить доступ к сущности либо по ее неполному имени, либо по псевдониму, если он указан.

Примеры:

- импортируем класс с полным именем `NameSpace1\NameSpace2\SomeClass`, после чего к нему можно будет обратиться по его сокращенному имени `SomeClass`:

  ```
  use NameSpace1\NameSpace2\SomeClass;
  $obj = new SomeClass();
  ```

- импортируем класс с полным именем `NameSpace1\NameSpace2\OtherClass`, указав для него псевдоним `OC`. После этого к нему можно будет обратиться по его псевдониму:

  ```
  use NameSpace1\NameSpace2\OtherClass as OC;
  $obj = new OC();
  use NameSpace1\NameSpace2;
  $result = NameSpace2\someFunc();
  ```

Мы не можем импортировать функцию, однако есть возможность импортировать пространство имен, в котором она объявлена. После этого можно вызвать эту функцию, обратившись к ней по неполному имени, в котором указано имя импортированного пространства имен.

Принципы написания программного кода PHP

Осталось рассмотреть основные принципы, согласно которым пишется код серверных программ, разрабатываемых на платформе PHP, неважно — с применением какого-либо фреймворка или без него.

Ранее упоминалось, что программное ядро платформы PHP позволяет включать фрагменты PHP-кода прямо в HTML-код страниц. В этом случае PHP-код будет обработан, а обычные теги HTML будут проигнорированы платформой и останутся неизменными.

PHP-код помещается внутрь тегов `<?php` и `?>`, которые носят общее название тега `php`. При этом в самом последнем выражении кода завершающий символ точки с запятой ставить не обязательно.

Примеры:

- этот код выведет на страницу строку "`PHP`":

```
<?php
  $s = "PHP";
  echo $s
?>
```

- этот код выведет ту же строку внутри тега абзаца:

```
<p><?php echo "PHP" ?></p>
```

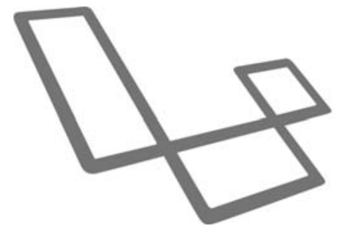
А код из листинга 24.14 выведет на страницу абзац, если переменная `n` хранит значение 1, и заголовок в противном случае.

Листинг 24.14

```
<?php if ($n == 1) { ?>
  <p>Абзац.</p>
<?php } else { ?>
  <h6>Заголовок</h6>
<?php } ?>
```

Если файл содержит только PHP-код, рекомендуется не указывать закрывающий тег `?>`. Это делается во избежание вывода на экран случайных символов пробела или перевода строки, которые могут оказаться за закрывающим тегом и способны нарушить дизайн Web-страниц.

Файлы с кодом PHP, независимо от того, включают ли они также и HTML-код, должны иметь расширение php.



ГЛАВА 25

Базы данных. Сервер данных MySQL

Любая программа, в том числе и серверная, в процессе функционирования обрабатывает какие-то данные, хранящиеся отдельно от программы. Так, в случае нашего сайта этими данными являются списки категорий, подкатегорий, статей, комментариев и пользователей, зарегистрированных на сайте, — серверные программы нашего сайта будут формировать на их основе разнообразные динамические страницы.

Эти данные удобнее всего хранить в *базах данных* — они уже включают в свой состав инструменты для структурирования информации и предоставляют средства для занесения новых данных, правки и удаления уже существующих и, разумеется, выборки данных, их поиска, фильтрации и сортировки.

Для хранения данных нашего сайта мы уже давно избрали *MySQL*. Это одна из наиболее популярных в настоящее время *СУБД* (система управления базами данных), часто используемая в серверном Web-программировании. Тому есть несколько причин: относительная компактность, нетребовательность к системным ресурсам, вполне достаточное быстродействие, богатые возможности и полная бесплатность.

Эта глава посвящена разговору о MySQL, поддерживаемом им формате баз данных и основных принципах, согласно которым хранятся эти данные.

Реляционные базы данных

Но сначала поговорим о самих базах данных. Точнее, о той их разновидности, которая носит название реляционной.

Введение в реляционные базы данных

В очень многих случаях обрабатываемые программами данные можно представить в виде таблицы или, если они слишком сложны для одной таблицы, — набора связанных таблиц. Таблицами представляются всевозможные описи, перечни, справочники, бухгалтерские книги. Даже если брать наш будущий сайт, то списки его

категорий, подкатегорий, статей и комментариев без труда могут быть представлены в табличном виде.

Каждая такая таблица имеет строго определенную структуру. В ней содержится определенное количество столбцов, а каждый столбец имеет строго определенное назначение. Например, список статей нашего сайта будет включать столбцы для хранения заголовка статьи, анонса, автора, содержания и подкатегории, к которой относится статья.

Из сказанного следует, что каждая строка таблицы представляет собой сложную сущность, хранящую в себе несколько значений разного типа. Мы, как программисты, можем считать эту строку аналогом экземпляра объекта JavaScript, а отдельный ее столбец — аналогом свойства этого экземпляра объекта. Каждая таблица может содержать произвольное количество строк.

Набор таблиц, записанный в файл или в группу файлов, называется *реляционной базой данных*. Это самая популярная на данный момент разновидность баз данных.

Отдельная строка таблицы в терминологии реляционных баз данных называется *записью*. А столбец, где сохраняется одно из составляющих отдельную запись значений, носит название *поля*.

Каждая таблица обязана иметь уникальное в пределах базы данных имя. По этому имени мы можем сослаться на эту таблицу, чтобы прочитать из нее данные или, наоборот, занести их в нее.

На рис. 25.1 показана схема подобной таблицы, хранящей список статей. Она включает пять полей, хранящих, соответственно, идентификатор статьи, ее заголовок, имя ее автора, подкатегорию и, наконец, само содержание. Как видим, в этой таблице сохранены четыре записи.

id	title	author	subcategory	content
3	Введение в HTML	Владимир Дронов	HTML	* * *
2	Структурирование текста	Владимир Дронов	HTML	* * *
1	Оформление текста	Владимир Дронов	HTML	* * *
4	Введение в CSS	Иван Иванов	CSS	* * *

Рис. 25.1. Схема таблицы — списка статей

Поля

Как мы уже знаем, поле таблицы хранит одно из значений, составляющих запись. Набор полей определяется при создании таблицы, но впоследствии может быть изменен.

Каждое поле имеет две важные характеристики:

- *имя* — используется, чтобы однозначно идентифицировать поле с целью извлечь из него значение или, наоборот, записать новое. Должно быть уникальным в пределах таблицы;

- *тип* — задает характер хранящегося в поле значения: будет оно строковым, числовым, логическим или представлять собой значение даты и времени.

Возьмем, к примеру, таблицу, показанную на рис. 25.1. Она включает пять полей:

- `id` — хранит уникальный идентификатор, или *номер*, записи и принадлежит к числовому типу;
- `title` — хранит заголовок статьи и принадлежит к строковому типу;
- `author` — имя автора статьи, строковый тип;
- `subcategory` — подкатегория, к которой принадлежит статья, строковый тип;
- `content` — содержимое статьи, строковый тип.

Имя и тип каждого поля также задаются при создании таблицы. Если нам потребуется изменить их, то придется править структуру таблицы.

Помимо имени и типа, описание поля может включать набор *атрибутов*, указывающих дополнительные параметры хранящихся в поле значений. Сюда можно отнести, в частности, атрибут, требующий обязательного занесения значения в поле (*обязательное поле*). А другой атрибут указывает для поля значение по умолчанию, которое будет занесено в поле при создании записи, если пользователь не занес туда никакого другого значения явно.

Отметим еще такой момент. Ранее мы описали тип каждого поля нашей пробной таблицы в терминологии языка программирования JavaScript, который уже неплохо знаем. Но типы полей, поддерживаемые реальными СУБД, практически никогда не соотносятся с типами данных, поддерживаемыми языками программирования (в том числе JavaScript и PHP), напрямую. Так, СУБД поддерживают несколько разновидностей числовых типов (целое число, число с плавающей точкой, денежная сумма и проч.) и как минимум два типа строковых полей (строка фиксированной длины и строка неограниченной длины). Некоторые форматы баз данных не предусматривают поддержку логического типа данных — вместо него рекомендуется применять либо числовой, либо строковый. Это всегда при разработке баз данных нужно иметь в виду.

Индексы и ключи

В процессе работы с данными, сохраненными в реляционных базах, очень часто приходится выполнять поиск нужной записи по значению ее поля (например, поля номера, что бывает чаще всего), а также фильтрацию и сортировку записей по заданным критериям. Эти операции выполняются весьма медленно и требуют большого объема оперативной памяти.

Давайте представим, что мы хотим найти в таблице статью по значению ее номера. Для этого СУБД придется прочитать довольно крупный фрагмент таблицы, переместить его в оперативную память и просмотреть на предмет наличия записи с указанным нами номером. И хорошо, если искомая запись будет найдена в первом же прочитанном из базы данных фрагменте таблицы... В противном же случае СУБД придется прочитать очередной фрагмент (опять же, большой и, опять же, целиком),

произвести поиск в нем, в случае неудачи прочитать третий фрагмент и т. д. Понятно, что длиться это будет долго...

Однако разработчики баз данных решили эту проблему достаточно давно, и решение получилось простое и очевидное. В самой базе данных создается массив, каждый элемент которого хранит значение поля, по которому наиболее часто выполняется поиск (скажем, поля номера), и указатель на запись таблицы, хранящую это значение. Этот массив для ускорения поиска отсортирован по значениям поля.

В таком случае СУБД уже не нужно просматривать всю таблицу в поисках искомой записи. Ей достаточно лишь прочитать в память этот массив, найти в нем элемент с указанным значением поля и загрузить из таблицы соответствующую ему запись. Понятно, что поиск будет выполнен много быстрее, поскольку, во-первых, массив относительно невелик по объему, а во-вторых, хранящиеся в нем данные отсортированы, что дополнительно — и весьма существенно! — сокращает время поиска.

Такой массив из значений поля и указателей на записи, откуда были взяты эти значения, называется *индексом*. А поле, на основе которого был создан индекс, носит название *индексированного*.

На рис. 25.2 схематически показана таблица списка статей с индексом, выполненным на основе поля номера (`id`). (Вообще, поле номера — первый кандидат на создание индекса.)

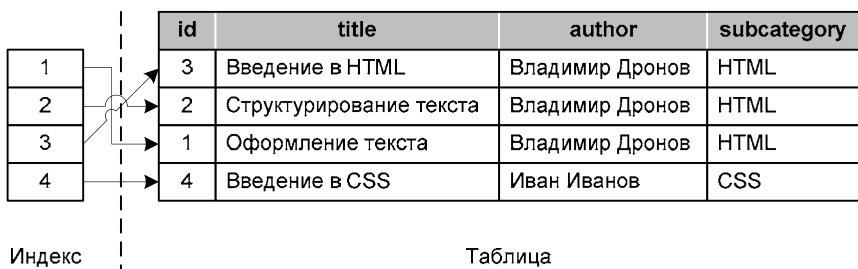


Рис. 25.2. Схема таблицы — списка статей с индексом

Индекс может быть создан на основе не одного, а нескольких полей. Однако такие индексы применяются нечасто.

Как и для полей, для индексов мы можем задать атрибуты. Наиболее часто используется атрибут, требующий уникальности занесенных в индексированное поле значений (*уникальный индекс*).

Из всех индексов, которые мы создаем для таблицы, один стоит несколько особняком. Это *ключевой индекс*, или *ключ*, который всегда создается на основе обязательного поля (т. е. поля, в которое обязательно должно быть занесено значение) и всегда является уникальным. А поле (или поля), участвующее в ключевом индексе, называется *ключевым*.

Ключевой индекс обеспечивает целостность данных, записанных в таблицу. Как правило, он создается на основе поля номера.

Как и таблицы и поля, индексы также часто имеют имена. Эти имена должны быть уникальными в пределах таблицы или базы данных — это зависит от конкретного формата, в котором хранится база.

Связи

Давайте еще раз посмотрим на таблицу, показанную на рис. 25.1. Что в ней настораживает?

То, что в поле подкатегории статьи (`subcategory`) хранится непосредственно ее название. Во-первых, оно, будучи строкой, занимает довольно много места в таблице. Во-вторых, если нам понадобится изменить название подкатегории, то придется выполнить поиск в таблице всех относящихся в ней статей и внести правки в соответствующие им записи, что отнимет много времени. В-третьих, нам, так или иначе, придется создавать таблицу списка подкатегорий, т. е. фактически дублировать названия подкатегорий сразу в двух таблицах.

Но мы можем хранить в поле `subcategory` не само название подкатегории, а номер соответствующей ей записи, хранящейся в таблице списка подкатегорий, как показано на рис. 25.3. Это позволит нам устранить все описанные проблемы.

id	title	author	subcategory
3	Введение в HTML	Владимир Дронов	1
2	Структурирование текста	Владимир Дронов	1
1	Оформление текста	Владимир Дронов	1
4	Введение в CSS	Иван Иванов	2

id	name
1	HTML
2	CSS

Рис. 25.3. Связанные таблицы списка статей и списка подкатегорий

Фактически мы создали *связь* между таблицами. Это связь типа *«один-ко-многим»*, где одна запись таблицы, называемой *первичной*, связана сразу с несколькими записями таблицы, которая называется *вторичной*. В нашем случае таблица списка подкатегорий будет первичной, а таблица статей — вторичной.

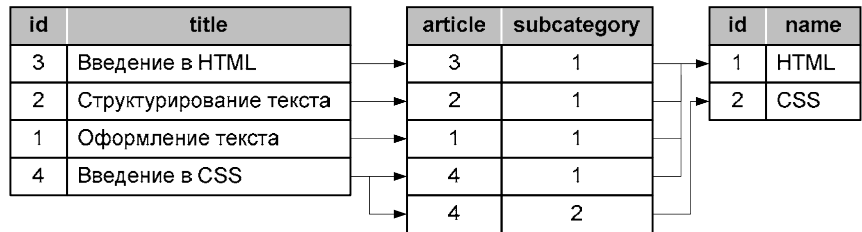
Также в качестве примера связи «один-ко-многим» можно привести связь между таблицами статей и комментариев. В этом случае таблица статей будет первичной, а таблица комментариев — вторичной. Следовательно, с одной статьей могут быть связаны сразу несколько комментариев.

На основе поля первичной таблицы, участвующего в образовании связи, должен быть создан ключевой индекс. А задействованное в связи поле вторичной таблицы также должно быть индексированным — индекс, созданный на основе этого поля, называется *внешним ключом*. Без этого связь создать не удастся.

Существуют и другие разновидности связей. Так, связь *«один-к-одному»* связывает одну запись первичной таблицы с одной записью вторичной. Например, если мы создадим таблицу списка зарегистрированных на нашем сайте пользователей и таблицу сведений о них (включающую, например, реальные имя и фамилию, пол, возраст и т. п.), то установим между ними связь именно такого вида. В самом деле, один пользователь должен иметь лишь одну связанную с ним запись сведений о пользователе — и только одну.

Связь *«многие-ко-многим»* позволяет связать несколько записей вторичной таблицы с несколькими записями первичной. С помощью подобной связи мы можем сделать так, чтобы одна статья могла присутствовать сразу в нескольких подкатегориях. Такая связь создается с применением еще одной таблицы, называемой *связующей*.

Схематически связь «многие-ко-многим» показана на рис. 25.4. Видно, что статья `Введение в CSS` связана сразу с двумя подкатегориями: `HTML` и `CSS`.



id	title
3	Введение в HTML
2	Структурирование текста
1	Оформление текста
4	Введение в CSS

article	subcategory
3	1
2	1
1	1
4	1
4	2

id	name
1	HTML
2	CSS

Рис. 25.4. Связь «многие-ко-многим» между таблицами списка статей и списка подкатегорий

Наиболее часто, впрочем, на практике применяются связи между таблицами вида «один-ко-многим». Прочие разновидности связей используются существенно реже и только в особых случаях.

Все СУБД сами следят за тем, чтобы связь между записями не нарушалась. Например, если мы попытаемся удалить подкатегорию `HTML`, то СУБД откажется это делать и выведет сообщение об ошибке, говорящее, что существуют записи вторичной таблицы, связанные с удаляемой. Однако мы можем изменить это поведение таким образом, чтобы при удалении записи первичной таблицы все связанные в ней записи вторичной таблицы удалялись автоматически. Это позволит нам быть уверенными, что, например, при удалении статьи все связанные с ней комментарии также будут удалены.

Язык SQL

Для выборки и манипулирования данными в современных СУБД используется специальный язык. Он носит название *SQL* (Structured Query Language, язык структурированных запросов).

На этом языке пишутся особые команды, называемые *запросами*. Каждый такой запрос выполняет одну операцию: выборку записей, добавление, правку или удаление записи, создание таблицы, индекса, базы данных и др.

Запросы SQL очень напоминают обычные предложения на английском языке. Давайте рассмотрим несколько примеров таких запросов:

- `select * from articles`

  Выбираем все поля всех записей таблицы `articles` (список статей). Символ звездочки (`*`) говорит о том, что должны быть выбраны все поля.

- `select name from subcategories`

  Выбираем лишь содержимое поля `name` (название) всех записей таблицы `subcategories` (список подкатегорий).

- `select id, name from subcategories order by name`

  Выбираем содержимое полей `id` (номер) и `name` всех записей таблицы списка подкатегорий, отсортировав их по полю `name`.

- `select * from articles where subcategory=1`

  Выбираем все поля только тех записей таблицы списка статей, в которых поле `subcategory` хранит значение 1 (т. е. статьи, относящиеся к подкатегории № 1 — `HTML`). Отметим, что в SQL оператор равенства обозначается одинарным символом =.

- ```
  select articles.title, subcategories.name from articles ➥
  inner join subcategories on articles.subcategory=subcategories.id
  ```

  Связываем таблицы списков статей и подкатегорий и извлекаем одновременно заголовки статей (`articles.title`) и названия подкатегорий (`subcategories.name`), к которым относятся эти статьи.

- `select count(*) from articles`

  Получаем количество записей таблицы `article`, или, говоря другими словами, количество всех опубликованных на сайте статей. Для этого мы используем *агрегатную* (то есть выполняющую какие-то вычисления на основе набора записей) функцию `count` языка SQL, которая возвращает количество записей в таблице.

- ```
  select subcategory, count(*) as cnt from articles group by ➥
  subcategory order by subcategory
  ```

  Получаем отсортированные номера подкатегорий и количество статей, относящихся к каждой из этих подкатегорий. Здесь, дабы указать СУБД, что требуется считать количество статей «внутри» подкатегорий, мы используем операцию группировки записей по подкатегориям. А для поля, в котором хранится количество статей, мы указываем псевдоним `cnt`, и теперь это поле будет доступно под этим псевдонимом.

- ```
  insert into subcategories (id, name) values ➥
  (3, "Всякая всячина")
  ```

  Добавляем в таблицу списка подкатегорий новую категорию `Всякая всячина` с номером `3`.
  ```

- `update subcategories set name="Прочее" where id=3`

  Меняем у вновь добавленной категории название на `Прочее`. Отметим, что мы ищем эту подкатегорию по ее номеру.

- `delete from subcategories where id=3`

  Удаляем только что исправленную категорию.

Если мы планируем разрабатывать сайты с применением какого-либо фреймворка, например Laravel, нам не понадобится писать запросы SQL самим, — достаточно лишь указать фреймворку выполнить определенную команду по обработке данных, вызвав нужный метод у нужного объекта, — и фреймворк сам сформирует SQL-запрос. Это особенно полезно в свете того досадного обстоятельства, что поддержка SQL в каждой СУБД реализована по-своему.

СУБД MySQL

Изучив особенности реляционных баз данных и кратко познакомившись с языком SQL, мы можем приступить к знакомству с СУБД MySQL, которую мы используем для хранения данных нашего сайта.

Типы данных, поддерживаемые MySQL

И начнем мы с того, что рассмотрим типы данных, поддерживаемые этой СУБД. Это необходимо для того, чтобы при создании полей таблиц выбрать для них наиболее подходящий тип.

Типы данных, поддерживаемые MySQL и применяемые на практике наиболее часто, приведены в табл. 25.1.

Таблица 25.1. *Типы данных, поддерживаемые MySQL*

Тип данных	Обозначение в MySQL	Примечание
Строки фиксированной длины	`VARCHAR`	От 1 до 255 символов. Длина строки задается при создании поля
	`CHAR`	От 1 до 255 символов. Длина строки задается при создании поля. Если строка имеет меньшую длину, она при сохранении будет дополнена пробелами
Строки произвольной длины	`TINYTEXT`	До 256 символов
	`TEXT`	До 65 535 символов
	`MEDIUMTEXT`	До 16 777 215 символов
	`LONGTEXT`	До 4 294 967 295 символов
Перечисление	`ENUM`	Хранит одно из заданных при создании поля строковых значений

Таблица 25.1 *(окончание)*

Тип данных	Обозначение в MySQL	Примечание
Целые числа	`TINYINT`	От –128 до 127 или от 0 до 255
	`SMALLINT`	От –32 768 до 32 767 или от 0 до 65 535
	`MEDIUMINT`	От –8 388 608 до 8 388 607 или от 0 до 16 777 215
	`INT`	От –2 147 483 648 до 2 147 483 647 или от 0 до 4 294 967 295
	`BIGINT`	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 или от 0 до 18 446 744 073 709 551 615
Числа с плавающей точкой	`FLOAT`	От $-3{,}402823466\times10^{38}$ до $-1{,}175494351\times10^{-38}$, 0 и от $1{,}175494351\times10^{-38}$ до $3{,}402823466\times10^{38}$
	`DOUBLE`	От $-1{,}7976931348623157\times10^{308}$ до $-2{,}2250738585072014\times10^{-308}$, 0 и от $2{,}2250738585072014\times10^{-308}$ до $1{,}7976931348623157\times10^{308}$
Числа с фиксированной точкой	`DECIMAL` и `NUMERIC`	Число фиксированной длины и с фиксированным положением десятичного разделителя, которые задаются при создании поля. Максимальная длина числа — 65 цифр
Логический	`BOOLEAN`	Логическая величина
Дата	`DATE`	Значения даты
Время	`TIME`	Значения времени
Дата и время	`DATETIME`	Значения даты и времени
	`TIMESTAMP`	Значения даты и времени, но несколько в другом формате
Двоичные данные фиксированной длины	`VARBINARY`	От 1 до 255 байтов. Длина значения задается при создании поля
	`BINARY`	От 1 до 255 байтов. Длина значения задается при создании поля. Если строка имеет меньшую длину, она при сохранении будет дополнена нулями слева
Двоичные данные произвольной длины	`TINYBLOB`	До 256 байтов
	`BLOB`	До 64 Кбайт
	`MEDIUMBLOB`	До 16 Мбайт
	`LONGBLOB`	До 4 Гбайт
Данные JSON	`JSON`	Данные в формате JSON

Строки фиксированной длины разработчики баз данных обычно называют просто строками. А строки произвольной длины носят название *memo*[1].

[1] Не запутайтесь: это английское слово и читается как ме́мо.

Преобразования типов данных MySQL в типы, поддерживаемые платформой PHP, выполняются фреймворком. Фреймворк также выполняет обратное преобразование.

Нужно иметь в виду еще один момент. Мы видим, что каждому типу данных PHP соответствует несколько аналогичных типов MySQL. Они отличаются лишь диапазоном значений, которые могут в них храниться. Но дело в том, что типы, позволяющие хранить строки большего размера и числа большего диапазона, занимают в базе данных больше места, чем типы, хранящие значения меньшего размера или диапазона. Например, поле типа `BIGINT` занимает больше места, чем поле типа `TINYINT`. Нам следует выбирать тот тип, который наилучшим образом подходит для хранения наших данных, — и не более.

Атрибуты полей и индексов

MySQL поддерживает несколько атрибутов, которые мы можем указать для полей таблиц:

- `NOT NULL` — указывает, что поле обязательно должно содержать значение (обязательное поле);
- `NULL` — указывает, что поле может хранить значение `null`;
- `DEFAULT` — задает для поля значение по умолчанию;
- `UNSIGNED` — превращает числовое поле в *беззнаковое*, позволяющее хранить лишь положительные числовые значения. Так, если обычное поле типа `TINYINT` позволяет хранить числа от –128 до 127, то поле типа `UNSIGNED TINYINT` уже может хранить числа от 0 до 255;
- `AUTO_INCREMENT` — указывает MySQL непосредственно при создании новой записи заносить в числовое поле целые числа, последовательно увеличивающиеся на единицу. Разработчики баз данных говорят, что в этом случае создается *поле счетчика*. Такие поля удобно использовать в качестве полей, хранящих номера записей, поскольку нам как программистам не придется самим заносить в них значения и следить за тем, чтобы они были уникальными.

Что касается индексов, то для них мы можем задать всего два атрибута:

- `UNIQUE` — создает уникальный индекс;
- `FULLTEXT` — создает на основе поля индекс наподобие тех, что формируются поисковыми сервисами. Такой индекс представляет собой список всех слов, встречающихся в тексте, с указанием, в каких записях присутствует то или иное слово. Применим лишь к полям типов `CHAR`, `VARCHAR` и `TEXT`.

Агрегатные функции

MySQL поддерживает следующие агрегатные функции:

- `count(`*<поле>*`)` — возвращает количество значений указанного *поля*, отличных от `null`. Если в качестве поля указать символ `*` (звездочка), возвращает общее количество записей;

- `min(<поле>)` — возвращает наименьшее из всех значений, хранящихся в указанном *поле*;
- `max(<поле>)` — возвращает наибольшее из всех значений, хранящихся в указанном *поле*;
- `sum(<поле>)` — возвращает сумму всех значений указанного *поля*;
- `avg(<поле>)` — возвращает арифметическое среднее, вычисленное на основе всех значений указанного *поля*.

Пользователи и их права

MySQL — это *сервер данных*, или, другими словами, *серверная СУБД*. Это значит, что она лишь предоставляет доступ к данным другим программам — клиентским, с которыми и работают пользователи. В нашем случае таким клиентом выступят серверные программы, составляющие в совокупности создаваемый нами динамический Web-сайт.

Другая особенность серверных СУБД — развитая система разграничения доступа на основе списка пользователей и их прав на доступ к данным (наподобие той, которая реализуется на сайтах и упоминалась в *главе 23*). Вот о ней-то мы сейчас и поговорим.

MySQL ведет список пользователей, имеющих права подключаться к управляемым им базам данных. Каждая позиция этого списка содержит имя пользователя («логин»), пароль, интернет-адрес компьютера, с которого допускается подключение к базе данных, и права пользователя. И если с именем пользователя и паролем все ясно, то остальные параметры требуют более обстоятельного разговора.

Интернет-адрес компьютера, с которого допускается подключение к базе данных, может представлять собой IP-адрес, доменное имя, обозначение `localhost`, указывающее на тот же компьютер, на котором установлен сам сервер данных (*локальный хост*), или символ `%`, говорящий, что этот пользователь может подключаться с любого компьютера, в том числе и с локального хоста.

Указание такого интернет-адреса поможет нам дополнительно обезопасить базы данных, управляемые сервером, от несанкционированного доступа. Скажем, указав для пользователя-администратора конкретный интернет-адрес подключения, мы лишим злоумышленников возможности подключиться к базе с любого другого компьютера, даже если они завладели именем и паролем администратора.

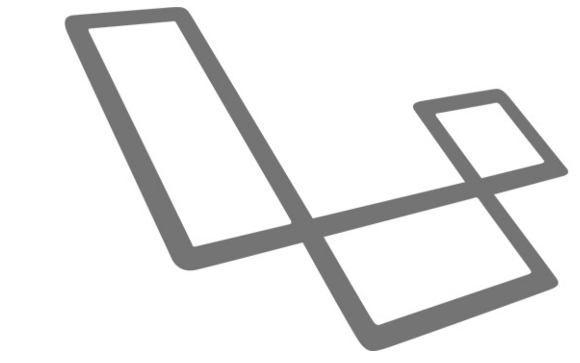
Что касается прав пользователя, то они обозначают операции, которые пользователь может выполнять с базами данных и хранящимися в них таблицами. Мы можем дать пользователю отдельные права лишь на выполнение чтения данных, на добавление, правку и удаление записей, на создание, правку и удаление таблиц, индексов и связей и, наконец, на создание и удаление баз данных. Мы можем дать разные права на доступ к разным базам данных или позволить подключаться лишь к одной базе из всех, что созданы на сервере. Мы даже имеем возможность дать пользователю права работать лишь с одной таблицей в одной-единственной базе данных.

Полные права на выполнение всех операций со всеми базами данных имеет лишь пользователь-суперадминистратор. Администратор базы или ее разработчик (часто эти обязанности выполняет один человек) имеет полные права лишь на доступ к базе, которую он, соответственно, администрирует или разрабатывает. А клиенты — в нашем случае это серверные программы, составляющие сайт, — имеют права лишь на чтение, добавление, правку и удаление записей в таблицах баз, где хранятся обрабатываемые ими данные.

Чтобы подключиться к базе данных, управляемой сервером MySQL, пользователю — человеку или программе — требуется указать имя и пароль. Сервер проверит, существует ли в списке пользователь с такими именем и паролем и может ли он подключаться с компьютера с таким интернет-адресом. Только если все эти условия выполнятся, сервер разрешит подключение.

А при выполнении любой операции сервер проверяет, имеет ли запрашивающий ее пользователь необходимые права. Только если пользователь их имеет, сервер выполняет запрошенную операцию.

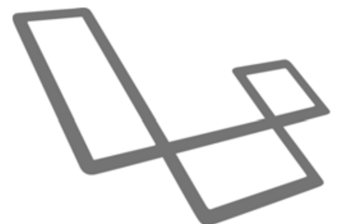
Более подробно работу с базами данных MySQL, в частности создание таблиц, индексов и связей и работа с пользователями, с помощью клиентской программы phpMyAdmin мы рассмотрим в *приложении 2*.



II. РАЗДЕЛ 5

Фреймворк Laravel

Глава 26.	Установка и настройка Laravel
Глава 27.	Миграции
Глава 28.	Модели
Глава 29.	Маршрутизация
Глава 30.	Контроллеры и действия
Глава 31.	Шаблоны
Глава 32.	Ввод и правка данных
Глава 33.	Разграничение доступа. Использование CAPTCHA
Глава 34.	Кэширование



ГЛАВА 26

Установка и настройка Laravel

Вот и настала пора заняться «сердцем» нашего будущего сайта — PHP-фреймворком Laravel. Начнем мы с того, что установим этот фреймворк в папку, где будут храниться все файлы разрабатываемого сайта, заодно создав там набор папок и файлов с изначальным программным кодом — той «печкой», от которой мы станем «плясать», — узнаем, что это за файлы и папки и зачем они нужны, и не забудем указать необходимые параметры, чтобы фреймворк, по крайней мере, смог подключиться к базе данных сайта.

В общем, мы выполним необходимые подготовительные действия перед началом собственно процесса разработки.

> ***Внимание!***
>
> Домашний сайт Laravel доступен по интернет-адресу **https://laravel.com/**. Там можно, в частности, найти всю документацию по этому программному продукту.

Программные требования Laravel

Перед тем как начать разработку сайта с применением Laravel, следует удостовериться, что программная платформа PHP, под которой он будет работать, удовлетворяет следующим требованиям:

- версия PHP — 5.6.4 или более новая;
- активны расширения OpenSSL, PDO, Mbstring, Tokenizer и XML.

В составе программного пакета хостинга OpenServer поставляется PHP версий 5.6 и даже 7.0, и нужную версию можно выбрать в диалоговом окне настроек этого пакета (см. *приложение 1*). Все необходимые PHP-расширения там уже активны.

Создание нового проекта

Нам нет необходимости отдельно загружать архив с фреймворком Laravel и распаковывать его в папку, где будут храниться файлы сайта, выполняя тем самым его установку. Мы можем просто указать создать там новый «пустой» сайт, основан-

ный на Laravel, а сам фреймворк и все дополнительные библиотеки, которые он использует в работе, будут автоматически загружены из Интернета и записаны в нужную папку.

Создание нового сайта, или, в терминологии Web-программирования, *проекта*, выполняется с помощью входящей в состав пакета OpenServer утилиты *Composer*. Это популярный в среде PHP-программистов установщик программных библиотек с функцией *разрешения зависимостей*, то есть установки всех библиотек, которые требует для работы устанавливаемая.

Чтобы воспользоваться Composer, нам понадобится запустить консоль OpenServer. Как это сделать, описано в *приложении 1*.

В окне консоли следует перейти в папку domains, в которую вложена папка localhost, где мы будем хранить файлы разрабатываемого сайта, воспользовавшись командой операционной системы `cd`:

```
cd domains
```

После чего набрать следующую строку:

```
composer create-project laravel/laravel <имя папки сайта> --prefer-dist
```

- Команда `create-project` указывает создать новый проект.
- `laravel/laravel` — это обозначение фреймворка, который, вместе с другими используемыми им библиотеками, нужно загрузить и установить.
- Параметр `--prefer-dist` указывает, по возможности, загружать и устанавливать окончательные редакции этих библиотек.

> **НА ЗАМЕТКУ**
>
> После установки пакет хостинга OpenServer создаст в папке *domains* папку *localhost* с файлами небольшого тестового сайта. Эта папка — неплохой кандидат на хранение файлов нашего сайта (только предварительно нужно удалить хранящиеся в ней файлы).

После нажатия клавиши <Enter> начнется процесс загрузки и установки библиотек. Продолжаться он будет довольно долго — минут десять (поскольку и сам Laravel, и все прочие библиотеки имеют немалый объем), и в его процессе Composer станет сообщать, какая библиотека в данный момент загружается.

Сигналом конца загрузки и установки библиотек послужит появление в окне консоли приглашения к вводу очередной команды с мигающим текстовым курсором.

После этого создадим в любом простейшем текстовом редакторе (подойдет и Блокнот, поставляемый в составе Windows) файл со следующим содержимым:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteRule ^(.*)$ public/$1 [L]
</IfModule>
```

Сохраним его под именем .htaccess (точка в начале имени обязательна!) в папке, где только что был создан новый проект. (В Блокноте при сохранении придется взять имя этого файла в кавычки.)

Теперь мы можем проверить только что созданный проект — заготовку для создания сайта — в действии. Запустим Web-обозреватель и выполним переход по интернет-адресу **http://localhost/**. После чего увидим страницу, показанную на рис. 26.1.

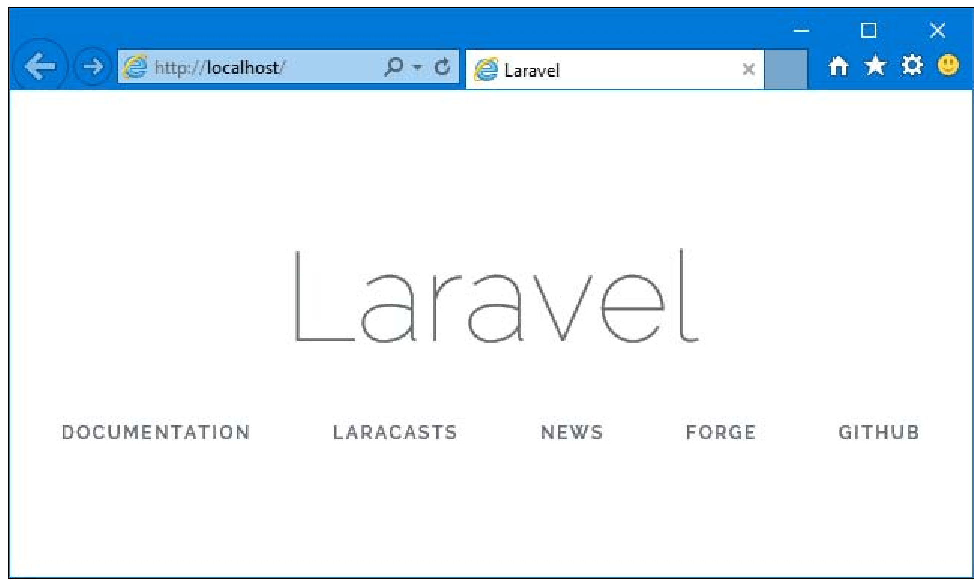


Рис. 26.1. Страница проекта, сформированная при его создании

Это страница проекта, которая формируется при его создании. Ничего интересного она не содержит — только название фреймворка и набор гиперссылок, по которым можно перейти в различные разделы его домашнего сайта. Она призвана лишь проверить, что вновь созданный проект работает.

Структура папок Laravel-проекта

Теперь давайте посмотрим, что хранится в папке, в которой мы создали новый проект (*папке проекта*).

Прежде всего, это три файла:

- artisan — утилита Artisan, используемая для прототипирования серверных программ различных типов (о прототипировании говорилось в *главе 23*) и выполнения служебных задач. Мы будем активно пользоваться этой утилитой в дальнейшем;
- .env — текстовый файл, хранящий базовые настройки проекта. Мы рассмотрим его чуть позже;
- readme.md — текстовый файл с информацией о Laravel. Содержит ссылки на раздел его домашнего сайта, где находится документация по фреймворку. В остальном не представляет какого-либо интереса.

Остальные файлы, находящиеся в этой папке, хранят служебную информацию, которой пользуется, в основном, лишь Composer.

Что касается папок, вложенных в папку проекта, то их довольно много. Давайте кратко рассмотрим их.

- app — хранит практически все серверные программы, выполняющие специфичные для сайта задачи. В частности, вложенная в нее папка Http/Controllers содержит файлы контроллеров.
- bootstrap — хранит серверные программы, выполняющие автоматическую загрузку других программ.

  Интересно, что часть этих программ генерируется самим фреймворком на основе сведений, заданных в файлах настроек. Такие программы хранятся в папке cache, вложенной в упомянутую папку.
- config — хранит основные файлы настроек. Они позволяют указать все параметры проекта, но, в отличие от файла базовых настроек, используются не очень часто.
- database — хранит серверные программы, относящиеся к работе с базами данных. Вложенная в нее папка migrations содержит файлы миграций.
- public — корневая папка сайта, созданного на основе Laravel. Все файлы, которые должны быть доступны посетителю как есть (графические изображения, таблицы стилей, внешние сценарии и проч.), должны сохраняться в ней.
- В папке public находятся следующие папки:
  - css — предназначена для хранения файлов с внешними таблицами стилей;
  - js — предназначена для хранения файлов сценариев;

  и следующие файлы:
  - index.php — серверная программа, запускающаяся при получении любого запроса (*стартовая программа*). В процессе работы выполняет все остальные серверные программы, входящие в состав сайта, — как предоставленные Laravel, так и написанные самим разработчиком;
  - .htaccess и web.config — задают параметры обработки сайта Web-серверами Apache HTTP Server и Microsoft Internet Information Services соответственно;
  - robots.txt — задает параметры обработки сайта поисковыми службами;
  - favicon.ico — значок сайта (пустой).

    Именно папка public должна быть сконфигурирована как корневая папка сайта, и только ее содержимое должно быть доступно в Интернете.
- resources — хранит файлы, не являющиеся программами как таковыми. В частности, во вложенной в нее папке views находятся файлы шаблонов.
- routes — хранит серверные программы, составляющие маршрутизатор.

- storage — служит для хранения файлов, либо выгруженных на сайт посетителями, либо сгенерированных самим фреймворком. Включает в себя следующие вложенные папки:
  - app — хранит файлы, выгруженные посетителем сайта, а также файлы, которые генерируются серверными программами, написанными разработчиком сайта;
  - framework — хранит файлы, сгенерированные фреймворком (файлы кэша, сессий и откомпилированных шаблонов);
  - logs — хранит файлы с журналами работы сайта.
- tests — хранит серверные программы, выполняющие автоматическое тестирование сайта. Мы не будем использовать автоматическое тестирование, так что эту папку можно сразу удалить.
- vendor — хранит все библиотеки, которые применяются для разработки сайта, в том числе и сам фреймворк Laravel.

Все упомянутые здесь папки и файлы необходимы для работы сайта. Поэтому ни в коем случае их нельзя удалять, иначе сайт может перестать работать. (Исключение представляет собой разве что папка tests, которую можно удалить сразу же.)

Настройка сайта

Перед тем как приступить собственно к разработке сайта, нам потребуется задать некоторые настройки. В противном случае мы даже не сможем работать с базой данных, выделенной под хранение внутренних данных сайта.

Настройки сайта производятся путем правки:

- файла .env, хранящегося в папке проекта, — он содержит базовые настройки: набор выражений, объявляющих различные переменные и задающих для них значения. Каждая такая переменная представляет один из базовых параметров проекта. Несмотря на отсутствие расширения php, это обычный PHP-файл;
- PHP-файлов, хранящихся в папке config папки проекта, — они содержат полный набор настроек.

  Каждый из этих файлов содержит код, возвращающий в качестве результата ассоциативный массив, элементами которого могут быть как обычные значения, так и другие ассоциативные массивы. В первом случае элемент этого массива представляет один из расширенных параметров сайта, во втором — целую группу параметров.

В большинстве случаев нам потребуется указать только базовые настройки. Однако о расширенных настройках тоже следует знать, поскольку без их правки не обойтись в некоторых случаях, в частности, при установке дополнительных библиотек.

Базовые настройки, как уже говорилось, хранятся в файле .env. Рассмотрим самые необходимые из них.

Настройка соединения с базой данных

Первое, что нам следует указать в настройках сайта, — это параметры соединения с базой данных, где будут храниться внутренние данные сайта.

> **ВНИМАНИЕ!**
>
> База данных, где будут храниться данные сайта, равно как и пользователь, от имени которого серверные программы станут к ней подключаться, уже должны быть созданы. Как это сделать, описано в *приложении 2*.

Далее приведены параметры, в которых указываются сведения о базе данных:

- `DB_CONNECTION` — формат используемой базы данных: `sqlite` (SQLite), `mysql` (MySQL — указан по умолчанию) и `pgsql` (PostgreSQL);
- `DB_HOST` — интернет-адрес компьютера, на котором установлен сервер данных (по умолчанию указан `127.0.0.1` — локальный хост, то есть этот же компьютер);
- `DB_PORT` — TCP-порт, используемый для взаимодействия с сервером данных: `3306` (порт MySQL — указан по умолчанию) или `5432` (порт PostgreSQL). Если сервер данных настроен на использование другого порта, следует указать именно его. Для взаимодействия с базами данных SQLite не требуется, так как базы данных такого формата не являются серверными;
- `DB_DATABASE` — имя базы данных;
- `DB_USERNAME` — имя пользователя, под которым серверные программы будут подключаться к базе данных;
- `DB_PASSWORD` — пароль этого пользователя.

Значения у всех этих параметров указываются без кавычек.

Пример указания параметров соединения с базой данных представлен в листинге 26.1.

Листинг 26.1

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=site
DB_USERNAME=siteuser
DB_PASSWORD=123456
```

Настройки отправки электронной почты

В *главе 33* мы будем изучать встроенную в Laravel подсистему разграничения доступа. Эта подсистема настолько развита, что может выполнять процедуру *сброса пароля*: посетитель заходит на страницу с формой, заносит в нее свой адрес электронной почты, после чего Laravel ищет в списке зарегистрированных пользователей пользователя с таким же адресом и отправляет ему электронное письмо.

В содержании письма имеется гиперссылка, ведущая на страницу, где посетитель сможет указать другой пароль вместо забытого им.

Чтобы в полной мере задействовать возможности этой подсистемы, нам понадобится задать для проекта сведения, необходимые для успешной отправки электронной почты. Задаются они следующими параметрами:

- `MAIL_DRIVER` — обозначение способа отправки писем. Нас более всего интересуют `smtp` (прямая отправка на SMTP-сервер) и `mail` (отправка средствами PHP). Вероятно, будет удобнее указать способ отправки `mail`;

  Дальнейшие параметры имеют смысл только в том случае, если выбран способ отправки `smtp`;
- `MAIL_HOST` — интернет-адрес используемого SMTP-сервера;
- `MAIL_PORT` — порт, используемый SMTP-сервером;
- `MAIL_USERNAME` — имя пользователя для подключения к SMTP-серверу;
- `MAIL_PASSWORD` — пароль пользователя;
- `MAIL_ENCRYPTION` — способ шифрования. Можно указать `null` (отсутствие шифрования), `ssl` и `tls`.

Вот пример указания отправки почты средствами PHP:

```
MAIL_DRIVER=mail
```

А в листинге 26.2 приведен пример настройки Laravel на отправку почты непосредственно SMTP-серверу.

Листинг 26.2

```
MAIL_DRIVER=smtp
MAIL_HOST=mail.supersite.ru
MAIL_PORT=465
MAIL_USERNAME=mail_robot
MAIL_PASSWORD=123456789
MAIL_ENCRYPTION=tls
```

Настройки режима работы сайта

Настройки режима работы сайта указывают, в каком виде будут выводиться на экран сообщения об ошибках, и еще некоторые сведения. Все необходимые параметры перечислены далее.

- `APP_DEBUG` — указывает, в каком виде будут выводиться сообщения об ошибках: `true` (максимально подробные сообщения, указано по умолчанию) или `false` (краткие сообщения).
- `APP_LOG_LEVEL` — указывает, насколько подробными будут сообщения об ошибках. Доступны значения: `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert` и

emergency — от более подробных сообщений к менее подробным. По умолчанию указано debug.

- APP_LOG — указывает разновидность создаваемых файлов журнала: single (один файл, указан по умолчанию), daily (по одному файлу на каждый день), syslog и errorlog (с применением одноименных форматов).

  Эта переменная изначально отсутствует в файле .env. Поэтому, если возникнет необходимость изменить параметр APP_LOG, ее объявление следует добавить в файл вручную.

Все эти три параметра уже имеют значения, наиболее подходящие для разработки сайтов. Однако перед публикацией сайта их требуется изменить (более подробно об этом будет рассказано в *главе 46*).

Прочие настройки

И наконец, рассмотрим еще пару настроек. Они не столь важны, как описанные ранее, и их значения обычно не изменяются, однако знать о них все же нужно.

- APP_URL — хранит интернет-адрес сайта. Используется утилитой Artisan при прототипировании серверных программ. Значение по умолчанию — **http://localhost** (локальный хост).
- APP_KEY — хранит секретный ключ в виде строки, представляющий собой случайный набор из 32 символов. Генерируется Composer при создании нового проекта и, как правило, не изменяется в дальнейшем.

Обработка ошибок

Когда мы начнем разрабатывать сайт, мы, скорее всего, столкнемся с большим количеством ошибок, допущенных в программном коде нами самими. Встретив такую ошибку, Laravel выведет страницу с ее описанием и указанием на файл и конкретное место в этом файле, где находится неправильно набранный код. Пример этой страницы показан на рис. 26.2.

Помимо этого, соответствующая запись заносится в файл журнала laravel.log, хранящийся в папке storage\logs, имеющейся в папке проекта. Вот пример такой записи, сообщающей о той же ошибке, что и показанная на рис. 26.2 страница:

```
[2016-12-02 10:55:00] local.ERROR: exception ⤷
'Symfony\Component\Debug\Exception\FatalErrorException' with message ⤷
'syntax error, unexpected 'view' (T_STRING)' in ⤷
C:\OpenServer\domains\localhost\routes\web.php:15
```

Здесь нас более всего интересует фрагмент, следующий за словами with message, взятый в апострофы и описывающий допущенную ошибку, и фрагмент после слова in, который указывает на программный файл с ошибкой.

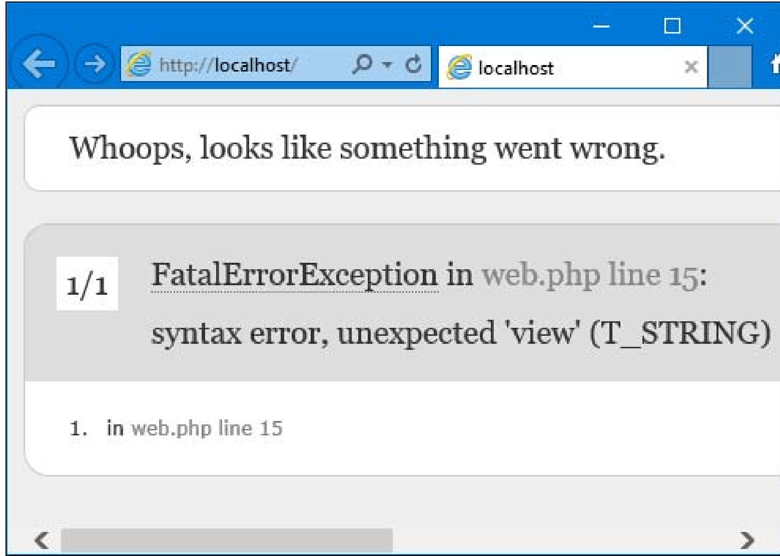
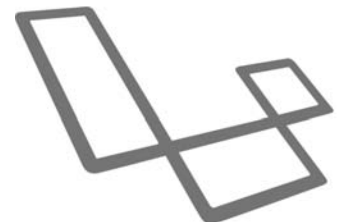


Рис. 26.2. Стандартная Web-страница Laravel с сообщением об ошибке



ГЛАВА 27

Миграции

В процессе разработки сайта нам придется создавать в базе данных всевозможные структуры: таблицы, поля, индексы и связи. Это можно сделать с применением специальных инструментальных средств наподобие популярной программы phpMyAdmin, описанной в *приложении 2* (и, кстати, также созданной на PHP). Если разрабатываемый сайт несложен, а разработка ведется одним человеком, так часто и делают.

А можно написать *миграцию* — особую серверную программу, которая сама создаст все необходимые таблицы, поля, индексы и связи. О миграциях уже упоминалось в *главе 23*, и настала пора поговорить о них подробнее.

Преимущества миграций

Если миграции поддерживаются в большинстве современных фреймворков (а так и есть на самом деле), стало быть, это кому-то нужно. Значит, они дают какие-то вполне ощутимые преимущества перед разработкой баз данных вручную.

- В разных форматах баз данных имеются довольно значительные различия в плане поддержки форматов полей, языковых особенностей SQL и т. п. При разработке баз данных вручную необходимо принимать это в расчет.

  Механизм миграции скрывает эти различия и позволяет писать программы, успешно создающие базы данных разных форматов.

- Часто приходится переносить изменения, сделанные в одной базе данных, в другие базы. Например, после добавления нового поля в таблицу базы данных на компьютере разработчика сайта следует добавить это же поле в рабочую базу. Часто это весьма трудоемко.

  Миграции позволяют выполнить подобного рода изменения без малейшего труда — достаточно лишь переписать файл с программой миграции на другой компьютер и запустить эту миграцию на выполнение.

- Иногда возникает необходимость сделать *откат*, то есть вернуть базы данных в состояние, предшествующее всем сделанным в ней изменениям. Если разра-

ботчик, сделавший эти изменения, не запомнил, что он делал, и не вел никаких записей, сделать это будет, пожалуй, невозможно…

Любая миграция предоставляет возможность отката, причем для ее реализации зачастую не придется писать никакого дополнительного кода.

Миграции в Laravel создаются очень просто. К тому же утилита Artisan поддерживает возможность прототипирования миграций.

Создание миграций

Миграция Laravel — это класс, являющийся потомком класса `Migration`. Файлы с классами миграций хранятся в папке database\migrations папки проекта.

Прототипирование миграций

Чтобы создать файл с начальным кодом миграции, следует открыть консоль Open-Server, перейти в папку проекта, использовав команду `cd`:

```
cd domains\localhost
```

после чего набрать команду вида:

```
php artisan make:migration <имя создаваемой миграции>
[--table=<имя таблицы, в которую будут вноситься изменения>]
[--create=<имя создаваемой таблицы>]
```

> ***Внимание!***
>
> Утилита Artisan написана на PHP. Поэтому она должна быть запущена под управлением исполняющей среды PHP, которая реализована в исполняемом файле *php.exe*. Имя этого файла указывается в команде перед именем утилиты Artisan, расширение *exe* может быть опущено.

Имя создаваемой миграции должно быть максимально понятным и включать описание выполняемых ей действий. Обычно оно состоит из нескольких слов, разделенных символами подчеркивания.

Если указано *имя таблицы, в которую будут вноситься изменения*, в код миграции будет подставлено выражение, получающее доступ к этой таблице. Это упростит нам написание кода, вносящего изменения в поля, индексы и связи этой таблицы.

Если указано *имя создаваемой таблицы*, в код миграции будет добавлено выражение, создающее эту таблицу.

После нажатия клавиши <Enter> запустится процесс создания новой миграции. По окончании ее создания будет выведено соответствующее сообщение.

Файл, в котором хранится вновь созданная миграция, получит имя вида: *<текущая дата, отдельные компоненты которой разделены символом подчеркивания>_<временная отметка>_<имя, указанное в вызове команды создания этой миграции>*.php.

В следующем примере мы переходим в папку созданного нами ранее проекта и отдаем команду на создание миграции. После ее выполнения у автора был создан файл миграции 2016_12_02_114626_create_categories_table.php:

```
cd domains\localhost
php artisan make:migration create_categories_table
```

Код новой миграции. Фасады Laravel

Программный код новой «пустой» миграции можно увидеть в листинге 27.1 (комментарии, вставленные Artisan, удалены).

Листинг 27.1

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

class CreateCategoriesTable extends Migration
{
  public function up()
  {
  }

  public function down()
  {
  }
}
```

Класс миграции получает имя, указанное в команде создания миграции, в котором каждое слово начинается с прописной буквы, а разделяющие слова символы подчеркивания удалены. Этот класс является потомком базового класса миграции `Migration`.

Класс миграции уже содержит два метода, не принимающие аргументов и не возвращающие результата: метод `up` выполняет все требуемые изменения в базе данных, а метод `down` производит откат.

Самые первые три выражения из листинга 27.1 выполняют импортирование необходимых классов. Класс `Migration` нам уже знаком, а два остальных — еще нет:

- `Schema` — предоставляет набор удобных статических методов для работы с таблицами, которые сами ничего не делают, а всего лишь вызывают методы других классов. Такие классы-«посредники», призванные упростить труд программиста, в терминологии Laravel называются *фасадами* (facades) — все они находятся в пространстве имен `Illuminate\Support\Facades`;
- `Blueprint` — представляет отдельную таблицу.

Создание структур данных

Поскольку наша база данных еще пуста, мы создадим в ней необходимые структуры данных: таблицы с полями, индексами и связями.

Создание таблиц

Для создания таблиц применяется статический метод `create` фасада `Schema`:

```
Schema::create(<имя таблицы>, <функция, создающая таблицу>)
```

Имя таблицы задается в виде строки. *Функция, создающая таблицу*, должна принимать единственный параметр — объект класса `Blueprint`, представляющий создаваемую таблицу. Результата метод `create` не возвращает.

Создание полей

Для создания полей применяется ряд методов класса `Blueprint`, описанных далее. Все они возвращают объект класса `Fluent`, представляющий созданное поле.

- `<таблица>->bigIncrements(<имя поля>)` — создает поле счетчика типа `BIGINT` (см. описание типов полей, поддерживаемых MySQL, в табл. 25.1).
- `<таблица>->increments(<имя поля>)` — создает поле счетчика типа `INT`.
- `<таблица>->mediumIncrements(<имя поля>)` — создает поле счетчика типа `MEDIUMINT`.
- `<таблица>->smallIncrements(<имя поля>)` — создает поле счетчика типа `SMALLINT`.

  Нужно отметить, что все поля счетчиков всех только что описанных типов являются беззнаковыми.
- `<таблица>->string(<имя поля>[, <длина поля>])` — создает поле типа `VARCHAR`. Если *длина поля* не указана, создается поле длиной 255 символов.
- `<таблица>->char(<имя поля>[, <длина поля>])` — создает поле типа `CHAR`. Если *длина поля* не указана, создается поле длиной 255 символов.
- `<таблица>->text(<имя поля>)` — создает поле типа `TEXT`.
- `<таблица>->mediumText(<имя поля>)` — создает поле типа `MEDIUMTEXT`.
- `<таблица>->longText(<имя поля>)` — создает поле типа `LONGTEXT`.
- `<таблица>->enum(<имя поля>, <массив значений>)` — создает поле типа `ENUM`, способное хранить одно из строковых значений, которые указаны в *массиве*.
- `<таблица>->tinyInteger(<имя поля>)` — создает поле типа `TINYINT`.
- `<таблица>->smallInteger(<имя поля>)` — создает поле типа `SMALLINT`.
- `<таблица>->mediumInteger(<имя поля>)` — создает поле типа `MEDIUMINT`.
- `<таблица>->integer(<имя поля>)` — создает поле типа `INT`.
- `<таблица>->bigInteger(<имя поля>)` — создает поле типа `BIGINT`.
- `<таблица>->unsignedTinyInteger(<имя поля>)` — создает беззнаковое поле типа `TINYINT`.

- *<таблица>*->unsignedSmallInteger(*<имя поля>*) — создает беззнаковое поле типа SMALLINT.
- *<таблица>*->unsignedMediumInteger(*<имя поля>*) — создает беззнаковое поле типа MEDIUMINT.
- *<таблица>*->unsignedInteger(*<имя поля>*) — создает беззнаковое поле типа INT.
- *<таблица>*->unsignedBigInteger(*<имя поля>*) — создает беззнаковое поле типа BIGINT.
- *<таблица>*->float(*<имя поля>*) — создает поле типа FLOAT.
- *<таблица>*->double(*<имя поля>*) — создает поле типа DOUBLE.
- *<таблица>*->decimal(*<имя поля>*[, *<длина числа>*[, *<количество цифр после десятичного разделителя>*]]) — создает поле типа DECIMAL. Если *длина числа* не указана, создается поле длиной в 8 цифр. Если не указано *количество цифр после десятичного разделителя*, оно будет установлено равным 2 цифрам.
- *<таблица>*->boolean(*<имя поля>*) — создает поле типа BOOLEAN.
- *<таблица>*->date(*<имя поля>*) — создает поле типа DATE.
- *<таблица>*->time(*<имя поля>*) — создает поле типа TIME.
- *<таблица>*->timeTz(*<имя поля>*) — создает поле типа TIME, также хранящее обозначение временной зоны.
- *<таблица>*->dateTime(*<имя поля>*) — создает поле типа DATETIME.
- *<таблица>*->dateTimeTz(*<имя поля>*) — создает поле типа DATETIME, также хранящее обозначение временной зоны.
- *<таблица>*->timestamp(*<имя поля>*) — создает поле типа TIMESTAMP.
- *<таблица>*->timestampTz(*<имя поля>*) — создает поле типа TIMESTAMP, также хранящее обозначение временной зоны.
- *<таблица>*->binary(*<имя поля>*) — создает поле типа BLOB.
- *<таблица>*->json(*<имя поля>*) — создает поле типа JSON.
- *<таблица>*->ipAddress(*<имя поля>*) — создает поле для хранения IP-адреса на основе одного из поддерживаемых типов.
- *<таблица>*->macAddress(*<имя поля>*) — создает поле для хранения MAC-адреса на основе одного из поддерживаемых типов.
- *<таблица>*->uuid(*<имя поля>*) — создает поле для хранения уникального идентификатора на основе одного из поддерживаемых типов данных.
- *<таблица>*->timestamps — создает поля created_at и updated_at, предназначенные для хранения даты создания и последнего изменения записи соответственно. Оба поля имеют тип TIMESTAMP.
- *<таблица>*->nullableTimestamps — то же, что и timestamps.

- *<таблица>*->timestampsTz — то же, что и timestamps, только предусматривает возможность хранить в создаваемых полях обозначения временной зоны.
- *<таблица>*->rememberToken — создает поле remember_token, предназначенное для хранения ключевой строки, которая применяется в процедуре восстановления пароля.

  Последние четыре метода не принимают аргументов.

Мы можем указать атрибуты для создаваемых полей. Для этого следует использовать описанные далее методы класса Fluent, которые вызываются непосредственно у нужного поля.

- *<поле>*->default(*<значение по умолчанию>*) — указывает для поля *значение по умолчанию*.
- *<поле>*->nullable — дает возможность хранить в поле значение null. Не принимает аргументов.
- *<поле>*->unsigned — делает поле беззнаковым. Не принимает аргументов.
- *<поле>*->first — помещает создаваемое поле в самое начало таблицы. Не принимает аргументов.
- *<поле>*->after(*<имя поля>*) — помещает создаваемое поле после поля с указанным *именем*.

В листинге 27.2 приведен полный код миграции, создающей в базе данных таблицу categories.

Листинг 27.2

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

class CreateCategoriesTable extends Migration {
  public function up() {
    Schema::create("categories", function(Blueprint $table) {
      $table->smallIncrements("id");
      $table->string("name", 40);
      $table->text("description");
      $table->smallInteger("order")->default(0);
      $table->timestamps();
    });
  }

  public function down() {
  }
}
```

В таблице `categories` создаются следующие поля:

- `id` — поле счетчика типа `SMALLINT` (для нумерации категорий этого хватит);
- `name` — строковое поле типа `VARCHAR` длиной 40 символов;
- `text` — memo-поле типа `TEXT`;
- `order` — целочисленное поле типа `SMALLINT` со значением по умолчанию 0;
- `created_at` и `updated_at` — поля типа `TIMESTAMP`.

Методы, задающие атрибуты для полей, в качестве результата возвращают тот же самый объект класса `Fluent`, у которого они были вызваны. Следовательно, мы можем задавать у любого поля сразу несколько атрибутов, соединяя вызовы этих методов, например:

```
$table->smallInteger("order")->default(0)->nullable()->first();
```

Создание индексов

Для создания индексов применяются методы класса `Blueprint`, описанные далее. Они вызываются у создаваемой таблицы.

- `index` — создает обычный индекс;
- `unique` — создает уникальный индекс;
- `primary` — создает ключ.

Все эти методы имеют одинаковый формат вызова:

```
<таблица>->index|unique|primary(<имя поля>|<массив с именами полей>
[, <имя индекса>])
```

Первым аргументом можно указать как строку с *именем поля* (тогда индекс будет создан на основе этого поля), так и *массив с именами полей*, также заданными в виде строк (тогда индекс будет создан на основе всех этих полей). Если не указано *имя индекса* (в виде строки), созданный индекс получит имя, сгенерированное самим Laravel.

> **ВНИМАНИЕ!**
>
> Методы `bigIncrements`, `increments`, `mediumIncrements` и `smallIncrements`, помимо создания самого поля счетчика, также создают на его основе ключ. Поэтому создавать его специально не нужно.

Листинг 27.3 показывает дополненный код метода `up` созданной ранее миграции, создающий, помимо полей, все необходимые индексы.

Листинг 27.3

```
class CreateCategoriesTable extends Migration {
  public function up() {
    Schema::create("categories", function(Blueprint $table) {
      $table->smallIncrements("id");
      $table->string("name", 40);
```

```
      $table->text("description");
      $table->smallInteger("order")->default(0);
      $table->timestamps();
      $table->unique("name");
      $table->index(["name", "order"]);
    });
  }
  . . .
}
```

Здесь создаются уникальный индекс на основе поля `name` и обычный индекс на основе полей `name` и `order`.

Методы, создающие индексы, также поддерживаются классом `Fluent`. Так что мы можем вызывать их у объектов этого класса, возвращенных методами, которые создают поля и их атрибуты. В этом случае никаких аргументов у методов не указывается.

В следующем примере мы создаем строковое поле `name` и сразу же — уникальный индекс на его основе:

```
$table->string("name", 40)->unique();
```

Создание связей

Процесс создания связи между таблицами выполняется в семь шагов.

1. Создается первичная таблица.
2. Создается миграция для вторичной таблицы.
3. В миграции для вторичной таблицы указывается создание внешнего ключа на основе поля, по которому будет устанавливаться связь. (Поскольку все поля счетчиков являются беззнаковыми, это поле также должно быть беззнаковым.) Для этого применяется метод `foreign` класса `Blueprint`, который вызывается так же, как методы, создающие индексы (см. ранее).

   Отметим, что метод `foreign` не поддерживается классом `Fluent`, поэтому все описанные здесь действия по созданию связи следует выполнять в отдельном выражении.
4. У объекта класса `Fluent`, возвращенного методом `foreign`, вызывается метод `references`:

   <поле>`->references(`*<поле первичной таблицы, по которому устанавливается связь>*`)`

   Поле первичной таблицы, по которому устанавливается связь, задается в виде строки.
5. У объекта класса `Fluent`, возвращенного методом `references`, вызывается метод `on`:

   <поле>`->on(`*<имя первичной таблицы>*`)`

   Имя первичной таблицы указывается в виде строки.

6. Если требуется указать, какие действия будут выполняться с записями вторичной таблицы при удалении связанной с ними записи первичной таблицы, следует вызвать у возвращенного методом `on` объекта класса `Fluent` метод `onDelete`:

   `<поле>->onDelete(<выполняемое действие>)`

   Выполняемое действие указывается в виде одной из строк:

   - `"cascade"` — записи вторичной таблицы, связанные с удаляемой записью первичной таблицы, также будут удалены;
   - `"set null"` — в связанные записи вторичной таблицы, в поле, по которому установлена связь, будут занесены значения `null`;
   - `"restrict"` — если удаляемая запись первичной таблицы связана с какими-то записями вторичной таблицы, удаление выполнено не будет;
   - `"no action"` — то же самое, что и `"restrict"`.

7. Если требуется указать, какие действия будут выполняться с полем внешнего ключа у записей вторичной таблицы при изменении значения ключевого поля в связанной записи первичной таблицы, следует вызвать у возвращенного методом `on` объекта класса `Fluent` метод `onUpdate`:

   `<поле>->onUpdate(<выполняемое действие>)`

   Выполняемое действие указывается в виде одной из строк:

   - `"cascade"` — значение поля внешнего ключа в записях вторичной таблицы также будет соответственно изменено;
   - `"set null"` — в поле внешнего ключа будут занесены значения `null`;
   - `"restrict"` — если исправляемая запись первичной таблицы связана с какими-то записями вторичной таблицы, правка значения ключевого поля в ней будет заблокирована;
   - `"no action"` — то же самое, что и `"restrict"`.

Листинг 27.4 показывает код метода `up` миграции, создающей в базе данных таблицу `subcategories` и связывающей ее с созданной ранее таблицей `categories`.

Листинг 27.4

```
class CreateSubcategoriesTable extends Migration {
  public function up() {
    Schema::create("subcategories", function (Blueprint $table) {
      // Создаем поля и индексы таблицы
      $table->smallIncrements("id");
      $table->string("name", 40)->unique();
      $table->text("description");
      $table->smallInteger("order")->default(0);
      $table->timestamps();
      $table->index(["name", "order"]);
```

```
      // Создаем поле, участвующее в образовании связи, и саму эту связь
      $table->smallInteger("category")->unsigned();
      $table->foreign("category")->references("id")->on("categories")
      ->onDelete("restrict")->onUpdate("restrict");
    });
  }
  . . .
}
```

В таблице `subcategories` создается поле `category`, которое связывается с ключевым полем `id` таблицы `categories`. При этом удаление записей первичной таблицы, имеющей связанные записи во вторичной таблице, запрещено, равно как и правка в них значения ключевого поля.

Правка и переименование структур данных

Перед тем как править уже существующие поля таблиц, следует установить дополнительную библиотеку doctrine/dbal, которая поможет Laravel выполнить правку максимально корректно, без потерь данных. Сделать это можно, набрав в консоли OpenServer команду:

```
composer require doctrine/dbal
```

и нажав клавишу <Enter>.

Добавление полей

Чтобы добавить новое поле в уже существующую таблицу, следует вызвать статический метод `table` фасада `Schema`. Он вызывается так же, как уже знакомый нам метод `create`, а вызов его ставится непосредственно в теле метода `up` или `down` класса миграции.

После этого в теле функции, передаваемой этому методу вторым аргументом, выполняется вызов метода, создающего нужное поле.

Пример кода миграции, создающего в таблице `subcategories` поле `hidden` логического типа, приведен в листинге 27.5.

Листинг 27.5

```
class ModifySubcategoriesTable extends Migration {
  public function up() {
    Schema::table("subcategories", function (Blueprint $table) {
      $table->boolean("hidden");
    });
  }
}
```

Правка и переименование полей

К сожалению, изменить тип какого-либо поля у нас получится далеко не всегда. Однако мы можем изменить длину строкового поля или поля типа DECIMAL, а также изменить их атрибуты.

Сначала в методе up или down класса миграции следует вызвать уже знакомый нам статический метод table фасада Schema. Далее в теле функции, передаваемой этому методу вторым аргументом, для каждого из исправляемых полей вызывается соответствующий метод, который укажет для поля требуемые параметры размера и атрибутов. После этого нужно будет всего лишь вызвать у результата, возвращенного этим методом, не принимающий аргументов и не возвращающий результата метод change.

Пример кода миграции, увеличивающей размер поля name таблицы subcategories до 50 символов, приведен в листинге 27.6.

Листинг 27.6

```
class ModifySubcategoriesTable extends Migration {
  public function up() {
    Schema::table("subcategories", function (Blueprint $table) {
      $table->string("name", 50)->change();
    });
  }
}
```

Переименовать поле можно вызовом метода rename класса Blueprint:

<таблица>->rename(*<старое имя поля>*, *<новое имя поля>*)

Этот метод вызывается там же — в теле функции, передаваемой вторым аргументом методу table.

В следующем примере мы переименовываем поле order в record_order:

```
$table->rename("order", "record_order");
```

Переименование таблиц

Переименовать таблицу можно вызовом статического метода rename фасада Schema. Он вызывается так же, как одноименный метод объекта Blueprint, непосредственно в теле метода up или down класса миграции.

В следующем примере мы даем таблицам categories и subcategories более короткие имена:

```
Schema::rename("categories", "cats");
Schema::rename("subcategories", "subs");
```

Удаление структур данных

Мы можем удалить связи, индексы, поля и целые таблицы, если они нам более не нужны.

Удаление связей

Удалить связь можно вызовом метода `dropForeign` класса `Blueprint`:

<таблица>`->dropForeign(`*<имя поля>*`|`*<массив с именами полей>*`)`

Этот метод вызывается у вторичной таблицы в теле функции, передаваемой вторым аргументом методу `table`. В качестве аргумента можно задать либо строку с именем поля, на основе которого создана удаляемая связь, либо массив с именами полей, указанных в виде строк, если нужно удалить сразу несколько связей.

В следующем примере мы удаляем из таблицы `subcategories` связь, созданную на основе поля `category`:

```
public function up() {
  Schema::table("subcategories", function (Blueprint $table) {
    $table->dropForeign("category");
  });
}
```

Удаление индексов

Для удаления индексов класс `Blueprint` предусматривает три метода:

- `dropIndex` — удаляет обычный индекс;
- `dropUnique` — удаляет уникальный индекс;
- `dropPrimary` — удаляет ключ.

Формат вызова этих методов одинаков:

<таблица>`->dropIndex|dropUnique|dropPrimary(`*<имя поля>*`|`*<массив с именами полей>*`)`

Эти методы вызываются в теле функции, передаваемой вторым аргументом методу `table`. В качестве аргумента можно задать либо строку с именем поля, на основе которого создан удаляемый индекс, либо массив с именами полей, указанных в виде строк, если индекс создан на основе нескольких полей.

В следующем примере мы удаляем индекс, созданный на основе полей `name` и `order`:

```
public function up() {
  Schema::table("subcategories", function (Blueprint $table) {
    $table->dropIndex(["name", "order"]);
  });
}
```

Удаление полей

Класс `Blueprint` поддерживает следующие методы, предназначенные для удаления полей из таблицы:

- `dropTimestamps` — удаляет поля `created_at` и `updated_at`;
- `dropTimestampsTz` — удаляет поля `created_at` и `updated_at` с возможностью хранения обозначения временной зоны;
- `dropColumn(`*<имя поля>*`|`*<массив с именами полей>*`)` — удаляет указанное поле или поля.

Эти методы вызываются в теле функции, передаваемой вторым аргументом методу `table`.

В следующем примере мы в методе `down`, который выполняет откат миграции, удаляем поля `description` и `order`:

```
public function down() {
  Schema::table("subcategories", function (Blueprint $table) {
    $table->dropColumn(["description", "order"]);
  });
}
```

Удаление таблиц

И, наконец, для удаления целой таблицы применяется статический метод `drop` фасада `Schema`:

`Schema::drop(`*<имя удаляемой таблицы>*`)`

Отметим, что, если таблица с заданным именем не существует, возникнет исключение.

В следующем примере мы удаляем таблицу `subcategories`:

```
public function down() {
  Schema::drop("subcategories");
}
```

Аналогичный метод `dropIfExists` выполняет удаление таблицы только в том случае, если она существует. Если же такой таблицы нет, метод ничего не делает.

Пример:

```
public function down() {
  Schema::dropIfExists("subcategories");
}
```

Выполнение и откат миграций

После того как миграция создана, ее следует выполнить. Для этого достаточно открыть консоль OpenServer, перейти в папку проекта и отдать команду (не забыв завершить ее нажатием клавиши <Enter>):

```
php artisan migrate
```

Отметим, что в этом случае будут выполнены все миграции, что хранятся в папке database\migrations и еще не выполнялись до этого момента.

> ***Внимание!***
>
> Для отслеживания, какие миграции уже были выполнены, Laravel создает в базе данных сайта таблицу `migrations`. Каждая запись этой таблицы соответствует одной успешно выполненной миграции. Если база данных пока еще находится в разработке, удалять эту таблицу не следует.

Сделать откат миграций можно отдачей команды:

```
php artisan migrate:rollback [--step=<количество откатываемых миграций>]
```

Если *количество откатываемых миграций* не указано, будет выполнен откат самой последней выполненной миграции.

Если нужно выполнить откат всех миграций, что имеются в папке database\migrations, нужно отдать команду:

```
php artisan migrate:reset
```

Для миграций, прошедших процедуру отката, Laravel удаляет соответствующие им записи в таблице `migrations` базы данных сайта. После этого мы сможем выполнить миграции, прошедшие процедуру отката, повторно.

Дополнительные возможности миграций

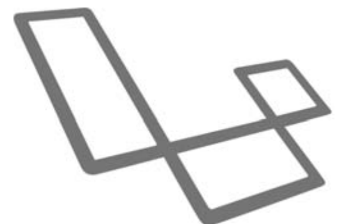
Осталось рассмотреть дополнительные возможности миграций, а именно — некоторые статические методы фасада `Schema`, позволяющие узнать, существует ли в базе данных указанная таблица или поле.

- `hasTable(`*<имя таблицы>*`)` — возвращает `true`, если таблица с указанным именем существует, и `false` в противном случае.
- `hasColumn(`*<имя таблицы>*`, `*<имя поля>*`)` — возвращает `true`, если поле с указанным именем существует в заданной таблице, и `false` в противном случае.
- `hasColumns(`*<имя таблицы>*`, `*<массив имен полей>*`)` — возвращает `true`, если все перечисленные в массиве поля существуют в заданной таблице, и `false` в противном случае.

Листинг 27.7 показывает пример кода, создающего таблицу `articles` и добавляющего в таблицу `categories` поле `hidden` только в том случае, если перечисленные ранее структуры отсутствуют.

Листинг 27.7

```
class ModifyTables extends Migration {
  public function up() {
    if (!(Schema::hasTable("articles"))) {
      Schema::create("articles", function (Blueprint $table) {
        . . .
      });
    }
    if (!(Schema::hasColumn("categories", "hidden"))) {
      Schema::table("categories", function (Blueprint $table) {
        $table->boolean("hidden");
      });
    }
  }
}
```



ГЛАВА 28

Модели

Создав в базе данных с помощью миграций все необходимые структуры (таблицы, поля, индексы и связи), можно приступать к разработке *моделей*. Модели будут использоваться для доступа к данным, хранящимся в этих таблицах, для их добавления, правки и удаления. Как и большая часть программ, входящих в состав разработанного на Laravel сайта, модели представляют собой классы.

Сразу нужно уяснить следующее:

- один объект класса модели представляет одну запись соответствующей таблицы в базе данных. Эта запись выбирается из таблицы на основе каких-либо критериев, например, ее номера;
- объект класса модели предоставляет инструменты как для получения значения полей, так и для занесения в них новых значений с целью сохранить в таблице;
- объект класса модели также позволяет добраться до связанных записей, если, конечно, между моделями установлена связь.

О создании связи между моделями мы поговорим в этой главе. А об инструментах, предназначенных для извлечения данных из модели и занесения в нее новых значений, разговор пойдет в последующих главах.

Модели Laravel: требования и соглашения

Сразу же имеет смысл поговорить о требованиях, предъявляемых фреймворком Laravel к классам моделей, и соглашениях по умолчанию, которым следуют модели. Это поможет нам прояснить некоторые моменты, о которых пойдет речь позже.

Начнем с требований к классам моделей. Их всего три:

- класс модели должен иметь имя, совпадающее с именем сущности, набором которых он манипулирует. Например, класс модели, манипулирующей категориями (записями таблицы `categories`), должен иметь имя `Category`, а класс модели, манипулирующей статьями (записями таблицы `articles`), — `Article`;

- класс модели должен быть потомком класса `Model` или его потомка. Этот класс предоставляет всю базовую функциональность модели, включая взаимодействие с базой данных, преобразование типов и проч.;
- код класса модели должен храниться в файле с именем, совпадающим с именем класса.

Модели Laravel следуют особым соглашениям по умолчанию, предполагая следующее:

- таблица базы данных, с которой взаимодействует модель, имеет имя, совпадающее с именем модели, но записанное во множественном числе. Так, модели `Category` должна соответствовать таблица `categories`, а модели `Article` — таблица `articles`;
- таблица, с которой взаимодействует модель, имеет служебные поля со следующими именами:
  - ключевое поле — `id`;
  - поле для хранения даты и времени создания записи — `created_at`;
  - поле для хранения даты и времени последнего изменения записи — `updated_at`;
- файлы с моделями находятся непосредственно в папке app папки проекта.

Это соглашения по умолчанию, и они не являются обязательными к исполнению. Мы можем указать для модели другую таблицу, другие имена служебных полей (как это сделать, будет рассказано далее) и поместить ее файл в другую папку.

Позже в этой главе, рассматривая способы установления связи между моделями, мы познакомимся с другими соглашениями по умолчанию, которым следуют модели Laravel.

Создание простых моделей

Создать простую модель, не содержащую связей с другими моделями и какой-либо дополнительной функциональности, очень просто. Скорее всего, нам даже не придется писать никакого кода вручную.

Прототипирование моделей. Базовый класс модели

Чтобы создать файл с классом модели, следует открыть консоль OpenServer, перейти в папку проекта и отдать команду такого вида:

```
php artisan make:model <имя класса модели> [--migration|--m]
```

Если указан командный ключ `--migration` или `--m`, помимо модели, будет создана «пустая» миграция. (О миграциях рассказывалось в *главе 27*.)

В результате будет создан базовый класс модели, наподобие класса `Category`, чей код можно увидеть в листинге 28.1.

Листинг 28.1

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
}
```

Как видим, здесь просто объявляется пространство имен `App` (в котором находится этот класс), импортируется базовый класс модели `Model` и на его основе создается сам класс модели (в нашем случае — `Category`). Никаких собственных свойств и методов этот класс изначально не содержит.

Задание параметров модели

В некоторых случаях нам понадобится задать различные параметры модели. Это может быть список полей, вовлекаемых в массовое присваивание, или, наоборот, не вовлекаемых в него, имя таблицы и имена служебных полей.

В *главе 31*, посвященной вводу данных, мы будем изучать массовое присваивание полученных от посетителя данных полям модели. Вероятно, это самый быстрый способ занести в модель новые данные.

Чтобы массовое присваивание успешно работало, нам следует указать, какие поля в него вовлечены. Вряд ли имеет смысл указывать все поля модели — так, например, не стоит вовлекать в массовое присваивание поле счетчика, где хранится номер записи, — все равно значение этого поля генерируется самой СУБД при создании новой записи.

Есть два способа указать, какие поля вовлечены в массовое присваивание:

- указать список полей, которые будут вовлечены в массовое присваивание (все остальные поля не будут вовлечены в него), присвоив массив с именами этих полей защищенному свойству `fillable` класса модели:

  ```
  class Category extends Model {
    protected $fillable = ["name", "description", "order"];
  }
  ```

  Здесь мы указываем, что в массовое присваивание будут вовлечены только поля `name`, `description` и `order`;

- указать список полей, которые, наоборот, не будут вовлечены в массовое присваивание (все остальные поля будут вовлечены в него), присвоив массив с именами этих полей защищенному свойству `guarded` класса модели:

  ```
  class Category extends Model {
    protected $guarded = ["id"];
  }
  ```

  Здесь мы указываем, что в массовое присваивание не будет вовлечено поле `id`.

Ранее мы узнали о соглашениях по умолчанию, которым следуют все модели Laravel. Но может случиться так, что существующие в базе данных структуры не удовлетворяют этим соглашениям. В этом случае нам могут помочь следующие защищенные свойства, поддерживаемые классами моделей:

- `table` — задает имя таблицы, с которой будет работать модель;
- `primaryKey` — задает имя ключевого поля;
- `timestamps` — указывает, будут ли при создании и удалении записи обновляться значения полей `created_at` и `updated_at`. Значение `true` задает их обновление (поведение по умолчанию), значение `false` предписывает не обновлять их.

  Если таблица не содержит полей `created_at` и `updated_at`, имеет смысл задать для свойства `timestamps` значение `false`.

И следующие константы класса:

- `CREATED_AT` — задает имя поля, где будет храниться дата и время создания записи;
- `UPDATED_AT` — задает имя поля, где будет храниться дата и время последнего обновления записи.

Пример использования этих свойств и констант показан в листинге 28.2.

Листинг 28.2

```
class Category extends Model {
  const CREATED_AT = "creation_date";
  const UPDATED_AT = "last_updated";
  protected $table = "cats";
  protected $primaryKey = "cat_id";
}
```

Создание связей

Если таблица, с которой работает модель, должна быть связана с другой таблицей, нам следует связать соответствующие модели явно.

Связь «один-ко-многим»

Сначала рассмотрим установление связи типа «один-ко-многим» как наиболее часто применяющуюся на практике. Она создается в четыре шага.

1. В классе модели, относящейся к первичной таблице (первичной модели), объявляется публичный метод, имя которого, как правило, совпадает с именем вторичной таблицы и который не должен принимать параметров. Этот метод создаст «прямую» связь — между первичной и вторичной моделями.

   Отметим, что имя этого метода задает *имя связи*. В дальнейшем, при работе с моделью, мы сможем извлечь связанные записи, обратившись к свойству мо-

дели, чье имя совпадает с именем связи. (О работе со связанными записями будет рассказано в *главе 30*.)

2. В теле объявленного на *шаге 1* метода вызывается метод `hasMany` класса `Model`:

```
<модель>->hasMany(<полное имя класса вторичной модели>
[, <имя поля внешнего ключа во вторичной таблице>
[, <имя связываемого поля первичной таблицы>]])
```

   Следует задать именно *полное имя класса вторичной модели* — с указанием пространства имен, в котором находится этот класс, и задать его в виде строки.

   При формировании имен полей, по которым будет выполнено связывание таблиц, Laravel руководствуется следующими соглашениями по умолчанию:

   - имя поля внешнего ключа во вторичной таблице имеет вид `<имя первичной таблицы>_id`, где имя первичной таблицы записано в единственном числе;
   - имя связываемого поле первичной таблицы — `id`.

   Так, если связываются модели `Category` и `Subcategory`, соответствующие таблицам `categories` и `subcategories`, предполагается, что во вторичной таблице `subcategories` есть поле внешнего ключа `category_id`, а в первичной таблице `categories` — связываемое поле `id`.

   Если же имена этих полей не удовлетворяют соглашением по умолчанию, мы можем указать *имя поля внешнего ключа во вторичной таблице* и *имя связываемого поля первичной таблицы* непосредственно в вызове метода `hasMany`.

   Результат, возвращенный методом `hasMany`, следует вернуть из метода, создающего связь, в качестве результата.

3. В классе модели, относящейся к вторичной таблице, объявляется публичный метод, имя которого, как правило, совпадает с именем первичной таблицы, записанном в единственном числе, и который не должен принимать параметров. Этот метод создаст «обратную» связь — между вторичной и первичной моделями.

   > **Внимание!**
   >
   > Если в таблице уже существует поле с таким же именем, следует дать этому методу другое имя.

4. В теле объявленного на *шаге 3* метода вызывается метод `belongsTo` класса `Model`:

```
<модель>->belongsTo(<полное имя класса первичной модели>
[, <имя поля внешнего ключа во вторичной таблице>
[, <имя связываемого поля первичной таблицы>]])
```

   Аргументы этого метода указываются в таком же виде, что и у метода `hasMany`.

   Результат, возвращенный методом `belongsTo`, следует вернуть из метода, создающего связь, в качестве результата.

В листинге 28.3 показан код моделей `Category` и `Subcategory`, между которыми установлена связь «один-ко-многим»: `Category` — первичная модель, а `Subcategory` —

вторичная. Предполагается, что имена полей таблиц, по которым устанавливается связь, удовлетворяют соглашениям по умолчанию: поле внешнего ключа вторичной таблицы имеет имя `category_id`, а связываемое поле первичной таблицы — имя `id`.

Листинг 28.3

```
class Category extends Model {
  public function subcategories() {
    return $this->hasMany("App\Subcategory");
  }
}

class Subcategory extends Model {
  public function category() {
    return $this->belongsTo("App\Category");
  }
}
```

В листинге 28.4 показан код тех же моделей `Category` и `Subcategory`, но уже для случая, если имена полей не удовлетворяют соглашениям по умолчанию. Здесь во вторичной таблице поле внешнего ключа называется `category`, а связываемое поле первичной таблицы — `cat_id`.

Листинг 28.4

```
class Category extends Model {
  public function subcategories() {
    return $this->hasMany("App\Subcategory", "category", "cat_id");
  }
}

class Subcategory extends Model {
  public function cat() {
    return $this->belongsTo("App\Category", "category", "cat_id");
  }
}
```

Отметим, что у метода, устанавливающего «обратную» связь вторичной модели `Subcategory` и первичной модели `Category`, мы задали имя `cat`, а не `category`. Дело в том, что в таблице `subcategories` уже есть поле `category`, и, если дать методу такое же имя, в дальнейшем при доступе к связанным записям возникнут проблемы.

Связь «один-к-одному»

Связь такого вида устанавливается так же, как связь «один-ко-многим», за двумя исключениями:

- имя метода первичной модели, задающего «прямую» связь, набирается в единственном числе;
- вместо метода `hasMany` вызывается метод `hasOne`.

Листинг 28.5 представляет код моделей `User` и `UserInfo`. Между этими моделями установлена связь «один-к-одному» и предполагается, что имена полей обеих таблиц удовлетворяют соглашениям по умолчанию.

Листинг 28.5

```
class User extends Model {
  public function userinfo() {
     return $this->hasOne("App\UserInfo");
  }
}

class UserInfo extends Model {
  public function user() {
    return $this->belongsTo("App\User");
  }
}
```

Связь «многие-ко-многим»

Связь «многие-ко-многим» устанавливается несколько сложнее. Но шагов для ее создания потребуется выполнить также четыре.

1. В первой из связываемых моделей объявляется публичный метод, имя которого, как правило, совпадает с именем второй связываемой таблицы и который не должен принимать параметров. Этот метод создаст связь.
2. В теле объявленного на *шаге 1* метода вызывается метод `belongsToMany` класса `Model`:

   ```
   <модель>->belongsToMany(<полное имя класса второй связываемой модели>
   [, <имя связующей таблицы>
   [, <имя поля внешнего ключа в первой связываемой таблице>
   [, <имя поля внешнего ключа во второй связываемой таблице>]]])
   ```

   При установлении связи такого вида Laravel руководствуется следующими соглашениями по умолчанию:

   - имя связующей таблицы имеет вид *<имя первой связываемой таблицы>_<имя второй связываемой таблицы>*, где имена обеих таблиц записываются в единственном числе и выстраиваются в алфавитном порядке;

- имя поля внешнего ключа в первой связываемой таблице имеет вид `<`*имя второй связываемой таблицы*`>_id`, где имя таблицы записано в единственном числе;
- имя поля внешнего ключа во второй связываемой таблице имеет вид `<`*имя первой связываемой таблицы*`>_id`, где имя таблицы записано в единственном числе;
- имена участвующих в установлении связи полей связующей таблицы совпадают с именами внешних ключей связываемых таблиц.

Например, если связываются модели `Article` и `Tag`, соответствующие таблицам `articles` и `tags`, то, если следовать соглашениям, связующая таблица должна иметь имя `article_tag` и содержать поля `article_id` и `tag_id`, поле внешнего ключа первой таблицы должно иметь имя `tag_id`, а аналогичное поле второй таблицы — `article_id`.

Если же имена связующей таблицы и полей не удовлетворяют этим соглашениям, мы можем указать их в вызове метода `belongsToMany`.

Результат, возвращенный методом `belongsToMany`, следует вернуть из метода, создающего связь, в качестве результата.

3. Во второй из связываемых моделей объявляется не принимающий параметров публичный метод. Имя этого метода обычно совпадает с именем класса первой связываемой модели, также записанным во множественном числе. Этот метод создаст связь.
4. В объявленном на *шаге 3* методе вызывается тот же самый метод `belongsToMany`, в котором на этот раз указывается всего один аргумент — полное имя класса первой связываемой модели. (Остальные параметры устанавливаемой связи будут взяты из вызова того же метода в первой связываемой модели.)

Пример установления связи между моделями `Article` и `Tag` можно увидеть в листинге 28.6. Предполагается, что имена связующей таблицы и полей удовлетворяют соглашениям: связующая таблица имеет имя `article_tag`, содержит поля `article_id` и `tag_id`, поле внешнего ключа первой связываемой таблицы называется `tag_id`, а поле второй таблицы — `article_id`.

Листинг 28.6

```
class Article extends Model {
  public function tags() {
    return $this->belongsToMany("App\Tag");
  }
}

class Tag extends Model {
  public function articles() {
```

```
    return $this->belongsToMany("App\Article");
  }
}
```

А пример из листинга 28.7 показывает установление связи между теми же моделями в случае, если связующая таблица имеет имя `tag_article`, включает поля `article` и `tag`, поле внешнего ключа первой связываемой таблицы называется `tag`, а поле второй таблицы — `article`.

Листинг 28.7

```
class Article extends Model {
  public function tags() {
    return $this->belongsToMany("App\Tags", "tag_article",
    "tag", "article");
  }
}

class Tag extends Model {
  public function articles() {
    return $this->belongsToMany("App\Article");
  }
}
```

Сквозная связь

Предположим, в базе данных есть три таблицы, последовательно связанные друг с другом: таблица `categories` связана с таблицей `subcategories`, а та — с таблицей `articles`. Мы можем создать в модели `Category` связь с моделью `Article` посредством модели `Subcategory` (которую можно назвать *промежуточной*), то есть создать *сквозную связь*.

Сквозная связь создается очень просто — всего за два шага:

1. В классе модели, относящейся к первичной таблице, объявляется публичный метод, имя которого, как правило, совпадает с именем вторичной таблицы и который не должен принимать параметров. Этот метод создаст сквозную связь.
2. В теле объявленного на *шаге 1* метода вызывается метод `hasManyThrough` класса `Model`:

```
<модель>->hasManyThrough(<полное имя класса вторичной модели>,
<полное имя класса промежуточной модели>
[, <имя поля внешнего ключа в промежуточной таблице>
[, <имя поля внешнего ключа во вторичной таблице>
[, <имя связываемого поля первичной таблицы>]]])
```

Соглашения по умолчанию требуют, чтобы:

- поле внешнего ключа в промежуточной таблице имело имя вида *<имя первичной таблицы>*`_id`, где имя таблицы записано в единственном числе;
- поле внешнего ключа во вторичной таблице имело имя вида *<имя промежуточной таблицы>*`_id`, где имя таблицы записано в единственном числе.

Так, если связываются модели `Category` и `Article` посредством модели `Subcategory`, предполагается, что во вторичной таблице `articles` есть поле внешнего ключа `subcategory_id`, в промежуточной таблице `subcategories` — поле внешнего ключа `category_id`, а в первичной таблице `categories` — связываемое поле `id`.

Если же имена этих полей не удовлетворяют соглашением по умолчанию, мы можем указать их имена в вызове метода `hasManyThrough`.

Результат, возвращенный методом `hasManyThrough`, следует вернуть из метода, создающего связь, в качестве результата.

Листинг 28.8 показывает пример кода, связывающего модели `Category` и `Article` через модель `Subcategory`. Предполагается, что все имена полей удовлетворяют соглашениям по умолчанию.

Листинг 28.8

```
class Category extends Model {
  public function articles() {
    return $this->hasManyThrough("App\Article", "App\Subcategory");
  }
}
```

А в листинге 28.9 можно увидеть код, связывающий те же самые модели. Однако здесь имена полей уже не удовлетворяют соглашениям по умолчанию: имя поля внешнего ключа во вторичной таблице `articles` — `subcategory`, имя поля внешнего ключа промежуточной таблицы `subcategories` — `category`, а имя связываемого поля первичной таблицы `categories` — `cat_id`.

Листинг 28.9

```
class Category extends Model {
  public function articles() {
    return $this->hasManyThrough("App\Article", "App\Subcategory",
    "category", "subcategory", "cat_id");
  }
}
```

Расширение функциональности модели

И, наконец, Laravel предоставляет нам возможность расширить функциональность моделей, создав у них вычисляемые поля и обработчики событий. Также мы можем объявить в классе модели дополнительные свойства и использовать их для хранения каких-либо данных.

Создание вычисляемых полей

Если нам необходимо часто выполнять какую-либо обработку получаемого из определенного поля модели значения (как их получать, мы узнаем в *главе 30*) и, возможно, также обрабатывать это значение перед занесением в него, будет проще создать *вычисляемое поле*.

Сделать это несложно. Достаточно объявить в классе модели, по крайней мере, один из двух перечисленных далее методов:

- `get`*<имя поля>*`Attribute` — вызывается при попытке извлечения значения из вычисляемого поля. Не принимает параметров. Должен возвращать результат, который и станет значением этого поля.

  Если этот метод отсутствует, к вычисляемому полю будет невозможно обратиться для получения значения;
- `set`*<имя поля>*`Attribute(`*<заносимое в вычисляемое поле значение>*`)` — вызывается при попытке занести значение в вычисляемое поле. В качестве параметра принимает значение, которое требуется занести в поле. Не должен возвращать результата.

  Если этот метод отсутствует, в вычисляемое поле невозможно будет занести новое значение.

Имя поля в обоих случаях записывается с прописной буквы. Если оно содержит несколько слов, разделенных символами подчеркивания, то каждое из этих слов записывается с прописной буквы, а символы подчеркивания удаляются.

Класс модели `Model` поддерживает свойство `attributes`. Оно хранит ассоциативный массив, ключи элементов которого являются именами полей модели, а значения этих элементов — суть значения, хранящиеся в соответствующих полях. Свойство `attributes` можно использовать, чтобы извлечь значение какого-либо поля модели, равно как и чтобы занести в поле модели обработанное значение.

Листинг 28.10 показывает класс модели `Category` с вычисляемым полем `name_order`. Извлекаемым из него значением станет значение поля `name`, к которому добавлено значение поля `order`, взятое в скобки. Отметим, что занести в это поле новое значение невозможно.

Листинг 28.10

```
class Category extends Model {
  public function getNameOrderAttribute() {
```

```
    return $this->attributes["name"] . "(" . $this->attributes["order"] .
    ")";
  }
}
```

Указание другого поля для поиска при внедрении модели

В *главе 29*, рассматривая средства маршрутизации Laravel, мы познакомимся с внедрением модели в контроллер. Если коротко, то, если в составе интернет-адреса будет указан какой-либо параметр, фреймворк может извлечь его, выполнить в таблице базы данных поиск записи на основании значения извлеченного параметра и передать созданный на основе этой записи объект класса модели в контроллер для дальнейшей обработки.

По умолчанию поиск записи выполняется по полю счетчика (`id`) — ищется запись, в которой значение этого поля равно значению параметра, взятого из интернет-адреса. Но мы можем указать другое поле для поиска записи. Для этого следует переопределить в классе модели публичный метод `getRouteKeyName` и в его коде вернуть имя нужного поля в виде строки в качестве результата.

Пример:

```
class Category extends Model {
  . . .
  public function getRouteKeyName() {
    return "slug";
  }
}
```

Теперь при внедрении модели поиск записи в таблице будет выполняться по полю `slug`.

Создание обработчиков событий

Часто бывает нужно выполнить какие-либо действия перед созданием, сохранением, удалением записи или после этого. Наилучший способ сделать это — создать обработчик соответствующего события.

Все поддерживаемые Laravel события приведены в табл. 28.1.

Таблица 28.1. *События, поддерживаемые моделями Laravel*

Событие	Описание
`creating`	Возникает перед сохранением новой записи
`created`	Возникает после сохранения новой записи
`updating`	Возникает перед обновлением уже существующей записи

Таблица 28.1 *(окончание)*

Событие	**Описание**
`updated`	Возникает после обновления уже существующей записи
`saving`	Возникает перед сохранением любой записи — вновь созданной или уже существующей
`saved`	Возникает после сохранения любой записи — вновь созданной или уже существующей
`deleting`	Возникает перед удалением записи
`deleted`	Возникает после удаления записи
`restoring`	Возникает перед извлечением записи из таблицы в модель
`restored`	Возникает после извлечения записи из таблицы в модель

Обработчики событий `creating`, `updating`, `saving`, `deleting` и `restoring` могут возвращать результат в виде логической величины. Если это `true`, соответствующая операция сохранения, удаления или извлечения записи успешно выполнится. Если же это `false`, операция будет отменена. Такую возможность можно использовать для выполнения каких-либо проверок.

Привязать обработчик к нужному событию нужной модели можно двумя способами. К сожалению, оба способа нельзя назвать удобными: придется править совершенно другой файл, а при использовании второго способа — еще и создавать новый.

Первый способ — запись кода обработчика в файл AppServiceProvider.php, находящийся в папке app\Providers. Этот файл хранит код класса `AppServiceProvider` — одного из провайдеров. (*Провайдером* в Laravel называется класс, входящий в состав программного ядра фреймворка и реализующий часть его ключевой функциональности.)

Обработчик события записывается в публичном методе `boot` провайдера `AppServiceProvider` в вызове одного из статических методов класса модели:

```
<модель>::<метод привязки обработчика к событию>(<функция-обработчик
события>)
```

Имя *`метода привязки обработчика к событию`* должно совпадать с именем события, к которому выполняется привязка обработчика (см. табл. 28.1). *`Функция-обработчик события`* должна принимать в качестве параметра объект модели, хранящий обрабатываемую — сохраняемую, удаляемую или извлекаемую — запись.

Пример кода, выполняющего привязку обработчика к событию `creating` модели `Category`, приведен в листинге 28.11.

Листинг 28.11

```
use App\Category;
class AppServiceProvider extends ServiceProvider {
  public function boot() {
    Category::creating(function ($data) {
      // Выполняются какие-то предварительные действия перед
      // сохранением
    });
  }
  . . .
}
```

Этот способ хорош в том случае, если нам нужно обрабатывать всего одно событие в какой-либо модели.

Второй способ рекомендуется применять, если требуется обрабатывать в модели сразу несколько событий. Он заключается в том, что весь код обработчиков выносится в отдельный класс, называемый *наблюдателем* (observer).

Класс наблюдателя — самый обычный, порождать его от какого-либо другого класса не нужно. Файл с его объявлением может быть сохранен в любой папке, при этом следует также поместить его в соответствующее пространство имен. Сами разработчики Laravel рекомендуют сохранять наблюдатели в папке app\Observers и помещать в пространство имен `App\Observers`.

Обработчик события в этом случае реализуется в виде одного из публичных методов класса-наблюдателя. Имя этого метода должно совпадать с именем обрабатываемого события (см. табл. 28.1). В качестве единственного параметра метод должен принимать объект модели, хранящий обрабатываемую в данный момент запись.

Листинг 28.12 показывает код объявления наблюдателя `CategoryObserver`. Файл с кодом его объявления должен быть помещен в папку app\Observers. В самом наблюдателе выполняется обработка событий `creating` и `created`.

Листинг 28.12

```
<?php
namespace App\Observers;
use App\Category;

class CategoryObserver {
  public function creating(Category $category) {
    if ($category->check()) {
      return true;
    } else {
      return false;
    }
  }
```

```
  public function created(Category $category) {
    // Выполняется какая-то дополнительная обработка
  }
}
```

В методе `creating` — обработчике одноименного события вызывается метод `check`, объявленный в классе модели `Category` самим разработчиком, этот метод может выполнять какую-либо проверку занесенных в модель значений. Если он вернет в качестве результата `true`, метод `creating` наблюдателя также вернет `true`, и создание записи будет выполнено, после чего выполнится метод `created` — обработчик события `created`. Если же метод `check` вернет `false`, метод `creating` также вернет `false`, и запись не будет создана.

Создав класс-наблюдатель, мы зарегистрируем его во фреймворке. Для этого следует в коде метода `boot` уже знакомого нам класса `AppServiceProvider` вызвать статический метод `observe` класса `Model`:

```
<модель>::observe(<полное имя класса наблюдателя>)
```

Полное имя класса наблюдателя указывается в виде строки. Отметим, что полное имя класса в строковом виде можно получить, обратившись к статическому свойству `class`, поддерживаемому всеми классами PHP.

Пример:

```
use App\Observers\CategoryObserver;
class AppServiceProvider extends ServiceProvider {
  public function boot() {
    Category::observe(CategoryObserver::class);
  }
  . . .
}
```

Произвольные свойства и методы модели

И наконец, ничто не мешает нам объявить в классе модели произвольные свойства и методы, как обычные, так и статические. Это могут быть свойства, хранящие вспомогательные данные, — например, названия полей таблицы, предназначенные для вывода на экран, методы, выполняющие проверку занесенных в поля значений по какому-либо сложному алгоритму (наподобие метода `check` из примера, показанного в листинге 28.12), и др.

Пример объявления дополнительного свойства и метода в классе модели `Category` показан в листинге 28.13.

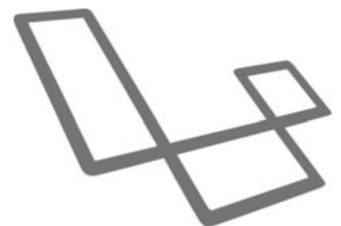
Листинг 28.13

```
class Category extends Model {
  public static $titles = [
    "name" => "Название",
```

```
    "description" => "Описание",
    "order" => "Порядок"
  ];

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

Здесь объявлено публичное статическое свойство `titles` и публичный метод `check`. Метод выполняет какую-либо проверку занесенных в поля модели значений, а свойство хранит названия полей модели в виде ассоциативного массива, в котором ключи элементов совпадают с именами полей, а значения элементов задают для них названия.



ГЛАВА 29

Маршрутизация

Теперь самое время познакомиться с маршрутизацией и инструментами Laravel, служащими для ее организации. Ведь, прежде чем какое-либо действие контроллера будет вызвано, фреймворк должен «узнать», какой контроллер и какое его действие следует выполнить при получении интернет-адреса в составе запроса.

Маршрутизацией в любом фреймворке, не только в Laravel, «заведует» особый программный модуль, называемый *маршрутизатором*.

Введение в маршрутизацию

Если бы мы не использовали для разработки сайта никаких фреймворков, а писали бы программный код на «голом» PHP, маршрутизация, собственно, вообще не была бы нам нужна. Все выполнял бы за нас Web-сервер.

Так, например, получив в составе запроса интернет-адрес (обозначение протокола, интернет-адрес хоста и номер порта опущены):

/php/categories/index.php

он запустил бы на выполнение серверную программу index.php, хранящуюся в папке php\categories корневой папки сайта.

Если серверная программа требует каких-то данных, переданных от посетителя (например, номер записи для поиска), эти данные передавались бы через GET-параметры. И, получив в составе запроса интернет-адрес:

/php/categories/edit.php?id=2

Web-сервер запустит на выполнение программу php\categories\edit.php и передаст ей значение GET-параметра `id` — 2.

Однако интернет-адреса такого рода не являются дружественными к пользователю. В настоящее время имеют хождение интернет-адреса другого вида — указывающие на раздел сайта, на страницу этого раздела, которую следует получить в ответ, или на действие, которое следует выполнить, хранящие отправляемые серверной про-

грамме значения непосредственно внутри себя (а не в GET-параметрах) и фактически представляющие собой обращение к папке, а не к программному файлу.

Давайте рассмотрим несколько примеров интернет-адресов такого рода:

- **/categories**

  Этот интернет-адрес указывает на раздел категорий, на страницу их списка.

- **/categories/2**

  Этот интернет-адрес указывает на раздел категорий, на страницу, отображающую отдельную категорию. В данном случае, поскольку в составе интернет-адреса передано в качестве параметра число 2, на странице должна быть отображена категория с номером 2.

- **/categories/create**

  А этот интернет-адрес указывает на раздел категорий, на страницу создания новой категории.

Как видим, такие интернет-адреса более логичны и лучше запоминаются посетителем. Более того, они лучше обрабатываются поисковыми службами.

Но в таком случае нам нужно как-то связать каждый из таких интернет-адресов с действием определенного контроллера. И при этом извлечь из адреса переданный в его составе параметр (или параметры — их может быть несколько) и передать его значение действию.

Здесь нам поможет маршрутизатор. Он сам вызовет нужное действие нужного контроллера и сам передает ему значения параметров, извлеченных из интернет-адреса. Нам нужно лишь указать правильные настройки маршрутизации.

Где хранятся настройки маршрутизации?

Для хранения настроек маршрутизации служат два файла, находящиеся в папке routes папки проекта:

- web.php — основной файл. При обработке указанных в нем настроек будут активизированы программные модули Laravel, реализующие хранение данных на стороне сервера (необходимо для последующей организации разграничения доступа) и защиты от сетевых атак. Эти модули, также представляющие собой классы, выполняются после завершения работы маршрутизатора и перед запуском контроллера, вследствие чего носят имя *посредников* (middleware).

  Поскольку практически все сайты реализуют разграничение доступа и требуют защиты от сетевых атак, в большинстве случаев для указания настроек маршрутизации следует использовать именно этот файл;

- api.php — вспомогательный файл. При обработке указанных в нем настроек упомянутые ранее посредники не задействуются.

  Этот файл следует применять только в тех случаях, если в сайте по какой-то причине не требуется организовывать разграничение доступа и защиту от се-

тевых атак или если для организации всего этого будут применены другие средства.

Мы в процессе разработки сайта будем указывать настройки маршрутизации в файле web.php, поскольку станем создавать подсистему разграничения доступа, и защита от сетевых атак нам также не помешает.

Что касается файла api.php, то заданные в нем маршруты нам не понадобятся. Так что мы можем сразу же открыть его и удалить присутствующее в нем единственное выражение, задающее маршрут.

Указание маршрутов

Маршрутом в Laravel называется связь между форматом интернет-адреса и соответствующим ему действием контроллера.

Простые маршруты

Для задания маршрутов применяются следующие методы фасада `Route`:

- `get` — обозначает, что в запросе была применена отправка данных методом GET. Отметим при этом, что интернет-адрес совсем не обязательно может включать GET-параметры, поскольку даже простой запрос на загрузку страницы без передачи серверной программе каких бы то ни было данных также выполняется с применением метода GET;
- `post` — обозначает отправку данных методом POST. Обычно применяется при вызове действия, создающего новую запись в базе;
- `put` — обозначает отправку данных методом PUT. Обычно применяется при вызове действия, изменяющего уже существующую запись, занося в нее полученные от посетителя данные;
- `patch` — обозначает отправку данных методом PATCH. Обычно применяется в тех же случаях, что и метод PUT;
- `delete` — обозначает отправку данных методом DELETE. Обычно применяется при вызове действия, удаляющего существующую запись.

> ***Внимание!***
>
> Web-формы, как мы знаем из *главы 6*, поддерживают только методы отправки данных GET и POST. Однако мы можем имитировать в форме отправку данных по методу PUT, PATCH или DELETE — как это сделать, будет описано в *главе 31*.

Все эти методы имеют одинаковый формат вызова:

```
Route::<метод>(<формат интернет-адреса>, <контроллер и действие>)
```

`Формат интернет-адреса` указывается в виде строки. Соответствующие контроллер и действие также задаются в виде строки в формате *`<имя класса контроллера>@<имя метода-действия>`*.

Вот несколько примеров:

- `Route::get("/categories", "CategoryController@index");`

  Указываем при обращении к интернет-адресу **/categories** вызвать действие `index` контроллера `CategoryController`. Это действие выведет на экран страницу со списком категорий.

- `Route::get("/categories/create", "CategoryController@create");`

  Указываем при обращении к интернет-адресу **/categories/create** вызвать действие `create` контроллера `CategoryController`. Это действие выведет на экран страницу для создания новой категории.

- `Route::post("/articles", "ArticleController@save");`

  При отправке данных методом POST по интернет-адресу **/articles** будет вызвано действие `save` контроллера `ArticleController`. Оно сохранит вновь созданную или исправленную статью.

Если в нашей практике данные с применением разных методов будут отправляться по одному и тому же интернет-адресу, в результате чего станет вызываться одно и то же действие контроллера, мы можем указать одинаковый маршрут сразу для нескольких методов отправки данных. Для этого следует использовать статический метод `match` фасада `Route`:

```
Route::match(<методы>, <формат интернет-адреса>, <контроллер и действие>)
```

Методы задаются в виде массива строк. Остальные аргументы указываются в том же виде, что и у рассмотренных ранее методов.

В следующем примере при обращении к интернет-адресу **/articles** методом GET и при отправке по тому же адресу данных методом DELETE следует выполнить действие `process` контроллера `ArticleController`:

```
Route::match(["get", "delete"], "/articles",
"ArticleController@process");
```

А метод `any` фасада `Route` позволит указать одинаковый маршрут сразу для всех методов отправки данных:

```
Route::any(<формат интернет-адреса>, <контроллер и действие>)
```

Следующий пример показывает, что при отправке данных любым методом по интернет-адресу **/comments** всегда будет вызываться действие `processall` контроллера `CommentController`:

```
Route::any("/comments", "CommentController@processall");
```

Параметризованные маршруты

Как мы убедились в начале этой главы, очень часто в составе интернет-адресов указываются значения различных параметров. Как правило, это номер записи, которую следует извлечь из таблицы.

Чтобы получить такой параметр, следует написать *параметризованный маршрут*. В таком маршруте получаемый параметр обозначается именем, взятым в фигурные скобки.

В следующем примере при обращении по адресу **/categories/<*номер категории*>** будет вызвано действие `show` контроллера `CategoryController`, которое получит номер категории в качестве аргумента:

```
Route::get("/categories/{id}", "CategoryController@show");
```

Вот так может выглядеть объявление метода `edit` класса `CategoryController`, реализующего это действие:

```
class CategoryController extends Controller {
  public function edit($id) { . . . }
}
```

Видно, что в объявлении метода указан параметр `id`, которому при вызове метода будет присвоен номер категории.

В параметризованном маршруте можно указывать и необязательные параметры. Имя такого параметра должно завершаться вопросительным знаком. Пример:

```
Route::get("/categories/{id?}", "CategoryController@edit");
```

Теперь параметр номера категории является необязательным.

Понятно, что в классе контроллера, в объявлении метода-действия следует указать для соответствующего параметра значение по умолчанию:

```
class CategoryController extends Controller {
  public function edit($id = 1) { . . . }
}
```

Правила для значений параметров в параметризованных маршрутах

В параметризованных маршрутах мы можем указать правила, которым должны удовлетворять значения определенных в них параметров. Если значение параметра в интернет-адресе не удовлетворяет этому правилу, Laravel считает, что интернет-адрес не совпадает с маршрутом.

Все рассмотренные ранее статические методы фасада `Route` в качестве результата возвращают объект класса `Route`, представляющий созданный маршрут. Этот класс поддерживает метод `where`, с помощью которого и указывается правила для значений параметров:

```
<маршрут>->where(<имя параметра>, <правило для параметра> | <список
правил для параметров>)
```

> ***Внимание!***
>
> Фасад `Route` и класс `Route`, представляющий маршрут, — это разные классы, поскольку они находятся в разных пространствах имен.

Мы можем указать при вызове этого метода:

- либо два аргумента — *имя параметра* и *правило для него*, представленное регулярным выражением PHP. Оба аргумента указываются в виде строк;
- либо один аргумент — *список правил для параметров*, если таковых в маршруте несколько. Список должен представлять собой ассоциативный массив, в качестве ключей элементов которого выступают имена параметров, а в качестве значений элементов — правила, заданные также в виде регулярных выражений.

Метод `where` в качестве результата возвращает тот же объект класса `Route`, у которого был вызван.

Вот пара примеров:

- указываем, что значение параметра `id` в этом маршруте должно представлять собой число из, как минимум, двух цифр:

```
Route::get("/categories/{id}", "CategoryController@edit")
->where("id", "[0-9]{2,}");
```

- указываем, что значение параметра `article_id` должно представлять собой число из, как минимум, двух цифр, а значение параметра `comment_id` — число, содержащее хотя бы одну цифру:

```
Route::get("/articles/{article_id}/comments/{comment_id}",
"CommentController@get")
->where(["article_id" => "[0-9]{2,}", "comment_id" => "[0-9]+"]);
```

Если сразу в нескольких маршрутах у нас определен параметр с одним и тем же именем, удовлетворяющий одним и тем же правилам, мы можем указать для него правила на уровне всего фреймворка. Для этого следует открыть файл RouteServiceProvider.php, хранящийся в папке app\Providers и содержащий код провайдера `RouteServiceProvider`, и вставить в код его метода `boot` вызов метода `pattern` фасада `Route`:

```
Route::pattern(<имя параметра>, <правило для параметра>)
```

Если нужно указать правила сразу для нескольких подобного рода параметров, применяется метод `patterns` того же фасада:

```
Route->patterns(<список правил для параметров>)
```

Аргументы для обоих этих методов указываются в том же виде, что и для метода `where`, рассмотренного ранее.

Листинг 29.1 показывает часть объявления класса провайдера `RouteServiceProvider`, в котором указываются правила для параметров `category_id`, `article_id` и `comment_id`, применяемые для всех маршрутов.

Листинг 29.1

```
class RouteServiceProvider extends ServiceProvider {
  . . .
  public function boot() {
```

```
    parent::boot();
    Route::pattern("category_id", "[0-9]+");
    Route::patterns([
      "article_id" => "[0-9]{2,}", "comment_id" => "[0-9]+"
    ]);
  }
  . . .
}
```

Всем хорош фреймворк Laravel! Вот только нередко код, выполняющий задачи сходного плана, часто оказывается разбросанным по нескольким файлам…

Именованные маршруты

Очень полезная возможность Laravel — назначение маршруту имени, по которому на него можно будет сослаться впоследствии. Маршруты, имеющие имена, называются *именованными*.

Дать маршруту имя можно, вызвав у представляющего его объекта класса `Route`, возвращенного одним из методов фасада `Route`, метод `name`:

```
<маршрут>->name(<имя маршрута>)
```

Имя маршрута указывается в виде строки. В качестве результата метод возвращает тот же объект класса `Route`, у которого был вызван.

В следующем примере мы задаем для маршрута, связывающего интернет-адрес **/categories** и действие `index` контроллера `CategoryController`, имя `cats`:

```
Route::get("/categories", "CategoryController@index")->name("cats");
```

Зачем маршрутам нужны имена, мы узнаем в последующих главах. Пока что заметим, что, обратившись к маршруту по имени, проще сгенерировать соответствующий ему интернет-адрес.

Указание посредников для маршрутов

В начале этой главы мы познакомились с посредниками, которые выполняются после завершения работы маршрутизатора и перед запуском контроллера и обычно реализуют, так сказать, глобальные задачи, наподобие защиты от сетевых атак и организации разграничения доступа.

Мы имеем возможность указать дополнительный посредник, который должен быть запущен перед выполнением контроллера, заданного в маршруте. Для этого служит метод `middleware`, который вызывается у объекта класса `Route`, возвращенного одним из методов одноименного фасада:

```
<маршрут>->middleware(<название посредника>|<список названий посредников>);
```

Можно указать либо строку с *названием одного посредника*, либо *массив с названиями сразу нескольких посредников*, также представленными в виде строк. В ка-

честве результата метод возвращает тот же объект класса `Route`, у которого был вызван.

В следующем примере мы указываем для маршрута посредник `auth`, реализующий разграничение доступа. Теперь по интернет-адресу **/categories/create** сможет зайти только зарегистрированный и выполнивший вход на сайт пользователь:

```
Route::get("/categories/create", "CategoryController@create")
->middleware("auth");
```

Массовое создание маршрутов

Очень часто для какого-либо контроллера указывается сразу несколько маршрутов — по одному на каждое его действие. И часто эти маршруты однотипные: один — для действия, выводящего список позиций, другой — для действия, выводящего позицию, которую выбрал посетитель, третий — для действия, создающего новую позицию, и т. д. Записывать все эти маршруты раз за разом крайне утомительно.

Базовые средства для массового создания маршрутов

Laravel может создать сразу все маршруты такого рода для заданного контроллера (*массовое создание маршрутов*). Выполняется это вызовом метода `resource` фасада `Route`:

```
Route::resource(<начало интернет-адреса>, <имя контроллера>
[, <параметры создаваемых маршрутов>])
```

Первые два аргумента указываются в виде строк. *`Начало интернет-адреса`* не должно предваряться символом слеша. О третьем аргументе мы поговорим позже.

Каждый вызов метода `resource` создает семь маршрутов, приведенных в табл. 29.1. Обозначением *`адр.`* помечено начало интернет-адреса, указанное в вызове этого метода. Параметр же получит имя, представляющее собой начало интернет-адреса, которое записано в единственном числе, имя этого параметра обозначено как *`пар.`*.

Таблица 29.1. *Маршруты, создаваемые вызовом метода `resource` фасада `Route`*

Метод отправки данных	Интернет-адрес	Действие контроллера	Имя маршрута	Задача, выполняемая действием контроллера
`GET`	***/<адр.>***	`index`	*`<адр.>`*`.index`	Вывод списка позиций
`GET`	***/<адр.>*/create**	`create`	*`<адр.>`*`.create`	Вывод страницы с формой для создания новой позиции

Таблица 29.1 *(окончание)*

Метод отправки данных	Интернет-адрес	Действие контроллера	Имя маршрута	Задача, выполняемая действием контроллера
POST	***/<адр.>***	store	*<адр.>*.store	Сохранение новой позиции
GET	***/<адр.>/{<пар.>}***	show	*<адр.>*.show	Вывод позиции с номером пар.
GET	***/<адр.>/{<пар.>}/edit***	edit	*<адр.>*.edit	Вывод страницы с формой для правки позиции с номером пар.
PUT	***/<адр.>/{<пар.>}***	update	*<адр.>*.update	Сохранение исправленной позиции с номером пар.
PATCH				
DELETE	***/<адр.>/{<пар.>}***	destroy	*<адр.>*.destroy	Удаление позиции с номером пар.

Например, вызов метода:

```
Route::resource("articles", "ArticleController");
```

создаст набор следующих маршрутов:

```
Route::get("/articles", "ArticleController@index")
->name("articles.index");
Route::get("/articles/create", "ArticleController@create")
->name("articles.create");
Route::post("/articles", "ArticleController@store")
->name("articles.store");
Route::get("/articles/{article}", "ArticleController@show")
->name("articles.show");
Route::get("/articles/{article}/edit", "ArticleController@edit")
->name("articles.edit");
Route::put("/articles/{article}", "ArticleController@update")
->name("articles.update");
Route::patch("/articles/{article}", "ArticleController@update")
->name("articles.update");
Route::delete("/articles/{article}", "ArticleController@destroy")
->name("articles.destroy");
```

И все это мы создали лишь одним выражением!

Если нужно массово создать маршруты сразу для нескольких интернет-адресов и контроллеров, используется метод `patterns` фасада `Route`:

```
Route::patterns(<список интернет-адресов и контроллеров>)
```

Передаваемый этому методу *список* должен быть представлен в виде ассоциативного массива. Ключами его элементов выступят начала интернет-адресов, а значениями элементов — имена контроллеров.

Пример:

```
Route::patterns([
  "articles" => "ArticleController",
  "comments" => "CommentController"
]);
```

Дополнительные параметры массово создаваемых маршрутов

Метод `resource` фасада `Route` позволяет нам указать для массово создаваемых маршрутов дополнительные параметры. Они задаются в третьем аргументе этого метода, который представляет собой ассоциативный массив, ключи элементов которого должны совпадать с названиями параметров, а значения элементов, собственно, укажут значения этих параметров.

Вот доступные для указания дополнительные параметры массово создаваемых маршрутов:

- `only` — список действий контроллера, для которых будут созданы маршруты (для остальных действий таковые не будут созданы). Значение параметра должно представлять собой массив имен действий, заданных в виде строк;
- `except` — список действий контроллера, для которых, напротив, маршруты созданы не будут (для остальных действий таковые будут созданы). Значение параметра должно представлять собой массив имен действий, заданных в виде строк.

  Следует указывать либо параметр `only`, либо параметр `except`, но не оба вместе;
- `names` — список имен создаваемых маршрутов. Значение параметра должно представлять собой ассоциативный массив, ключами элементов которого станут имена действий, а их значениями — назначаемые соответствующим маршрутам имена;
- `parameters` — имя параметра для создаваемых маршрутов. Значение параметра должно представлять собой ассоциативный массив, ключом единственного элемента станет начало интернет-адреса, а его значением — имя параметра.

Пример кода, массово создающего маршруты и задающего для них дополнительные параметры, представлен в листинге 29.2.

Листинг 29.2

```
Route::resource("articles", "ArticleController", [
  "only" => ["index", "show"],
  "name" => ["index" => "articles.list", "show" => "articles.get"],
  "parameters" => ["articles" => "id"]
]);
```

Здесь указывается создание маршрутов только для действий `index` и `show` контроллера — для них задаются имена `articles.list` и `articles.get`, а для параметра, хранящего номер статьи, указывается имя `id`.

Внедрение модели в контроллер

Очень часто параметры, указываемые в интернет-адресах, представляют собой числа — номера записей. Получив такой номер в качестве аргумента, действие контроллера извлекает из нужной таблицы запись с указанным номером в виде объекта класса модели. Для этого применяется код, приведенный в листинге 29.3.

Листинг 29.3

```
// Код маршрута
Route::get("/articles/{id}", "ArticleController@show");

// Код контроллера
use App\Article;
class ArticleController extends Controller {
  public function show($id) {
    // Выполняется поиск статьи с полученным номером
    $article = Article::findOrFail($id);
    . . .
  }
}
```

Однако Laravel может предоставить действию контроллера не номер записи, а саму эту запись, отыскав ее в таблице по полученному в составе интернет-адреса номеру и оформив в виде готового к обработке объекта класса модели. Или, говоря другими словами, выполнить *внедрение модели* в контроллер.

Внедрение модели является частью более общего механизма Laravel, называемого *внедрением зависимостей*. Более подробно и на конкретных примерах мы рассмотрим его в последующих главах.

Неявное внедрение модели

Проще всего выполнить внедрение модели неявно. Для этого следует выполнить несколько простых условий:

- имя параметра в маршруте должно совпадать с именем соответствующего класса модели;
- имя параметра в объявлении метода-действия контроллера должно совпадать с именем параметра маршрута;
- для параметра в объявлении метода-действия контроллера должен быть указан тип данных — класс соответствующей модели.

Пример реализации неявного внедрения модели можно увидеть в листинге 29.4.

Листинг 29.4

```
// Код маршрута
Route::get("/articles/{article}", "ArticleController@show");

// Код контроллера
use App\Article;
class ArticleController extends Controller {
  public function show(Article $article) {
    // Параметр $article хранит объект класса Article, представляющий
    // найденную статью
    . . .
  }
}
```

У Laravel есть интересная особенность. Если программное ядро не найдет подходящую запись модели, он создаст новую запись этой же модели, которую и передаст в контроллер. Это можно использовать для создания новой записи.

По умолчанию Laravel ищет запись по значению ее поля `id`. Однако мы можем указать в классе модели другое поле для поиска, переопределив в нем метод `getRouteKeyName`. Как это сделать, было описано в *главе 28*.

Явное внедрение модели

Если мы по какой-то причине не можем соблюсти перечисленные ранее условия (например, не можем указать имя параметра маршрута, совпадающее с именем класса модели) или хотим реализовать свою логику поиска записи, нам придется выполнить явное внедрение модели.

В этом случае нам опять понадобится класс провайдера `RouteServiceProvider`, что хранится в одноименном файле в папке app\Providers. В его методе `boot` мы вызовем метод `model` фасада `Route`:

```
Route::model(<имя параметра>, <имя класса модели>)
```

Имя параметра и *имя класса модели* указываются в виде строк. Полное имя класса модели можно получить через статическое свойство `class`.

Пример кода, выполняющего явное внедрение модели, можно увидеть в листинге 29.5.

Листинг 29.5

```
use App\Article;
class RouteServiceProvider extends ServiceProvider {
  . . .
  public function boot() {
    parent::boot();
```

```
    Route::model("selected_article", Article::class);
  }
  . . .
}
```

После этого мы можем получать объект класса модели, хранящий найденную запись, через параметр `selected_article` (листинг 29.6).

Листинг 29.6

```
// Код маршрута
Route::get("/articles/{selected_article}", "ArticleController@show");

// Код контроллера
use App\Article;
class ArticleController extends Controller {
  public function show(Article $article) { . . . }
}
```

Может возникнуть потребность реализовать какой-либо другой алгоритм поиска записи в таблице, отличный от простого сравнения значения полученного из интернет-адреса параметра со значением, хранящимся в указанном поле таблицы. Для этого можно применить метод `bind` фасада `Route`:

`Route::bind(`*<имя параметра>*`, `*<функция, выполняющая поиск записи>*`)`

Имя параметра задается в виде строки. *Функция, выполняющая поиск записи*, должна принимать в качестве единственного параметра значение параметра, извлеченного из интернет-адреса, и возвращать найденную запись.

Вызов метода `bind` должен производиться также в коде метода `boot` класса `RouteServiceProvider`.

Листинг 29.7 показывает код, реализующий собственную логику поиска записи. Здесь выполняется поиск записи, поле `id` которой хранит значение, совпадающее со значением извлеченного из интернет-адреса параметра, а поле `published` хранит значение `true`.

Листинг 29.7

```
use App\Article;
class RouteServiceProvider extends ServiceProvider {
  . . .
  public function boot() {
    parent::boot();
    Route::bind("article", function ($id) {
      return Article::where("id", $id)->where("published", true)
```

```
      ->first();
    });
  }
  . . .
}
```

Группы маршрутов

Последняя возможность, предоставляемая Laravel в плане настройки маршрутизации, — объединение маршрутов в *группы* и указание особых параметров для всех маршрутов, что входят в такую группу.

Для создания группы применяется метод `group` фасада `Route`:

```
Route::group(<параметры маршрутов, входящих в группу>,
<функция, в теле которой создаются входящие в группу маршруты>)
```

`Параметры маршрутов` указываются в виде ассоциативного массива, ключи элементов которого являются названиями параметров, а значения элементов — значениями этих параметров. *`Функция, в теле которой создаются маршруты`*, не должна принимать параметров и возвращать результат, в ее теле должен присутствовать только код, создающий входящие в группу маршруты.

Параметры групп, которые, как мы помним, будут применены ко всем маршрутам, входящим в группу, приведены далее:

- `prefix` — префикс интернет-адреса. Если интернет-адреса в нескольких маршрутах начинаются с одинакового фрагмента, мы можем поместить эти маршруты в группу и указать этот фрагмент в качестве префикса в ее параметрах.

  Рассмотрим в качестве примера два маршрута, интернет-адреса в которых начинаются с фрагмента `"/categories"`:

  ```
  Route::get("/categories", "CategoryController@index");
  Route::get("/categories/{id}", "CategoryController@show");
  ```

  Мы можем объединить их в группу с префиксом:

  ```
  Route::group(["prefix" => "categories"], function () {
    Route::get("/", "CategoryController@index");
    Route::get("/{id}", "CategoryController@show");
  });
  ```

- `middleware` — посредник. Если несколько маршрутов должны быть обработаны с применением одного и того же посредника, мы можем поместить их в группу, указав этот посредник для всей группы.

  Например, эти два маршрута будут обработаны с применением посредника `auth`, реализующего разграничение доступа:

  ```
  Route::group(["middleware" => "auth"], function () {
    Route::get("/users/", "UserController@index");
    Route::get("/users/{id}", "UserController@show");
  });
  ```

- namespace — пространство имен для классов контроллеров. По умолчанию все классы контроллеров должны находиться в пространстве имен App\Http\Controllers, но мы можем создать в нем вложенное пространство имен и поместить в него контроллеры, выполняющие сходные задачи. (Так, кстати, поступают довольно часто.)

  Здесь нужно обязательно иметь в виду, что в качестве значения параметра namespace должно быть указано не полное имя пространства имен, а только тот его фрагмент, что находится после App\Http\Controllers.

  В качестве примера рассмотрим группу маршрутов, вызывающих действия контроллера, который находится в пространстве имен App\Http\Controllers\Admin:

```
// Код контроллера
namespace App\Http\Controllers\Admin;
class UserController extends Controller { . . . }

// Код маршрута
Route::group(["namespace" => "Admin"], function () {
  Route::get("/login/", "UserController@login");
  Route::get("/logout/", "UserController@logout");
});
```

Физические интернет-адреса

В процессе обработки интернет-адреса, полученного в составе запроса, Laravel сравнивает его со всеми маршрутами и, при достижении совпадения, запускает действие контроллера, указанное в совпавшем маршруте. Все просто.

Но что случится, если интернет-адрес не совпал ни с одним маршрутом, указанном в маршрутизаторе?

В этом случае Laravel предполагает, что интернет-адрес указывает на файл, хранящийся в папке public или в одной из вложенных в нее папок. Это может быть файл с графическим изображением, внешней таблицей стилей, внешними Web-сценариями или даже обычной статической Web-страницей. Такие интернет-адреса называют *физическими*.

Встретив интернет-адрес, не совпавший ни с одним из маршрутов, Laravel предполагает, что это физический интернет-адрес, и завершает свою работу, передавая его Web-серверу. Последний выполнит считывание файла, находящегося по этому адресу, и пересылку его Web-обозревателю, отправившему запрос.

Еще в *главе 26* говорилось, что корневой папкой сайта, разработанного с применением Laravel, является папка public, находящаяся в папке проекта. Именно в ней Web-сервер и будет искать файлы, соответствующие физическим интернет-адресам.

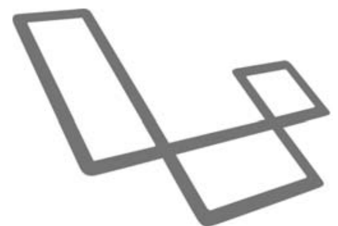
Рассмотрим в качестве примера такие маршруты:

```
Route::get("/categories", "CategoryController@index");
Route::get("/categories/{id}", "CategoryController@show");
```

Если в составе запроса получен интернет-адрес **/categories**, будет выполнено действие `index` контроллера `CategoryController`, а если получен интернет-адрес **/categories/1** — действие `show` того же контроллера, вдобавок это действие в качестве аргумента получит значение 1.

Но при получении интернет-адреса **/image.png**, который не совпадает с обоими маршрутами, будет выполнена отправка файла image.png, хранящегося в папке public. А если будет получен интернет-адрес **/styles/main.css**, Web-обозреватель получит файл main.css, находящийся в папке public\styles.

Laravel выполняет обработку физических интернет-адресов самостоятельно. Нам для этого ничего настраивать в нем не нужно.



ГЛАВА 30

Контроллеры и действия

Что ж, мы познакомились с моделями, посредством которых производится выборка данных из базы, и маршрутизацией, выполняющей нужное действие нужного контроллера в ответ на получение запроса с интернет-адресом определенного формата. Следующим вполне логичным шагом будет изучение контроллеров, которые, собственно, и выполняют всю работу по преобразованию хранящихся в базе данных в «понятные» Web-обозревателю страницы.

Контроллеры Laravel: требования и соглашения

Начнем, как и ранее, с требований, предъявляемых к контроллерам и действиям фреймворком Laravel.

- Класс контроллера должен иметь имя, совпадающее с именем сущности, набор которых он обрабатывает, а завершаться его имя должно словом `Controller`. Например, класс контроллера, манипулирующего категориями, должен иметь имя `CategoryController`, а класс контроллера, манипулирующего статьями, — `ArticleController`.
- Класс контроллера должен быть потомком класса `Controller` или `BaseController`.

  Класс `BaseController` является частью фреймворка Laravel. Он предоставляет минимальную функциональность контроллера, которой может хватить во многих случаях. Однако этот класс не позволяет реализовать разграничение доступа и проверку полученных от посетителя данных на корректность.

  Напротив, класс `Controller` является частью разрабатываемого проекта, как и созданные нами миграции, модели и маршрутизатор. Он использует несколько предоставляемых Laravel трейтов, добавляющих ему поддержку разграничения доступа и проверку полученных от посетителя данных. Мы можем дополнительно расширить функциональность этого класса, добавив ему нужные свойства и методы.

  Рекомендуется порождать новые классы контроллеров от класса `Controller`. Средства прототипирования Laravel так и делают.

- Классы контроллеров должны находиться в пространстве имен `app\Http\Controllers` или вложенном в него. В последнем случае можно поместить указывающие на эти контроллеры маршруты в группу и задать для нее это пространство имен в параметре `namespace` (за подробностями — к *главе 29*).
- Код класса контроллера должен храниться в файле с именем, совпадающим с именем класса.
- Все файлы контроллеров должны находиться в папке app\Http\Controllers или вложенной в нее папке, если используется вложенное пространство имен.
- Методы, реализующие действия, обязательно должны быть публичными.

Впрочем, если мы планируем создавать новые контроллеры с помощью инструментов прототипирования Laravel, последние сами позаботятся о том, чтобы выполнить эти требования.

Создание контроллеров

Чтобы создать средствами прототипирования файл с классом контроллера, следует открыть консоль OpenServer, перейти в папку проекта и отдать команду такого вида:

```
php artisan make:controller <имя класса контроллера> [--resource]
```

Если командный ключ `--resource` не указан, будет создан «пустой» контроллер, без действий. Если же этот командный ключ указать, контроллер будет содержать действия `index`, `create`, `store`, `show`, `edit`, `update` и `destroy`, и для контроллера можно сразу же выполнить массовое создание маршрутов (см. *главу 29*).

Базовый класс контроллера, создаваемый в результате этих действий в случае указания ключа `--resource`, показан в листинге 30.1. Созданный без указания этого ключа контроллер выглядит так же, за исключением отсутствия в нем каких бы то ни было методов.

Листинг 30.1

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CategoryController extends Controller
{
  public function index()
  {
  }

  public function create()
  {
  }
```

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

  public function show($id)
  {
  }

  public function edit($id)
  {
  }

  public function update(Request $request, $id)
  {
  }

  public function destroy($id)
  {
  }
}
```

Получение данных от посетителя

Получить данные от посетителя мы можем несколькими путями — в зависимости от того, в каком виде были приняты эти данные.

Проще всего получить значения параметров, переданных в составе интернет-адреса через параметризованные маршруты (об их создании говорилось в *главе 29*).

Если требуется просто извлечь значения таких параметров, мы запишем в объявлении метода-действия параметры, чьи имена совпадают с параметрами интернет-адреса. После чего мы сможем получить значения, просто обратившись к этим параметрам в теле метода-действия. Пример:

```
// Код маршрута
Route::get("/categories/{id}", "CategoryController@show");

// Код контроллера
public function show($id) {
  // Получаем значение параметра id, переданного в составе
  // интернет-адреса, через $id и обрабатываем его
}
```

Если значением какого-либо параметра из интернет-адреса является указатель на конкретную запись в таблице — обычно это номер или *слаг* (о слагах будет рассказано в *главе 40*), мы можем получить непосредственно объект соответствующей модели, хранящий эту запись. Для этого достаточно указать и в объявлении метода-

действия, и в маршруте параметр, чье имя совпадает с именем нужного класса модели, и задать для него в качестве типа тот же класс. Пример:

```
// Код маршрута
Route::get("/categories/{category}", "CategoryController@show");

// Код контроллера
use App\Category;
public function show(Category $category) {
  // Получаем модель, хранящую категорию с указанным номером, через
  // $category и используем ее в работе
}
```

Если же нужно получить данные, переданные методами GET или POST, например, из Web-формы, нам придется предварительно получить объект класса `Request`, представляющий запрос. Сделать это проще всего, указав в методе-действии параметр с типом `Request`:

```
use Illuminate\Http\Request;
public function store(Request $request) {
  . . .
}
```

В результате Laravel сам создаст объект класса `Request` и присвоит его этому параметру при вызове метода. Это, кстати, конкретный пример внедрения зависимостей, о котором упоминалось ранее.

Получить запрос также можно, вызвав функцию `request` без аргументов:

```
$request = request();
```

Класс `Request` поддерживает ряд свойств и методов, позволяющих нам получить эти данные.

Прежде всего, это свойства, чьи имена совпадают с именами GET- и POST-параметров, полученных в этом запросе. Вот пример их использования:

```
use Illuminate\Http\Request;
public function store(Request $request) {
  // Получаем имя, описание и порядок следования категории,
  // заданные посетителем в Web-форме
  $name = $request->name;
  $desc = $request->description;
  $order = $request->order;
}
```

Помимо этого, поддерживается ряд методов, позволяющих получить такие данные:

- `input(`*<имя параметра>*`[, `*<значение по умолчанию>*`])` — возвращает значение GET- или POST-параметра с указанным *именем*. Если таковой отсутствует, возвращается *значение по умолчанию*, либо, если таковое не указано, `null`. Пример:

  ```
  $order = $request->input("order", "0");
  ```

- `query(<имя параметра>[, <значение по умолчанию>])` — то же самое, что и `input`, но работает только с GET-параметрами;
- `all` — возвращает ассоциативный массив, ключи элементов которого представляют имена параметров, а значения элементов — значения этих параметров. Не принимает аргументов. Пример:

```
$data = $request->all();
$name = $data["name"];
```

- `only(<список параметров>)` — то же самое, что `all`, но возвращаемый ассоциативный массив включает лишь параметры, перечисленные в *списке*. Можно либо перечислить строковые имена параметров непосредственно в вызове метода, либо оформить их как массив. Пример:

```
$data = $request->only("name", "desc");
```

- `except(<список параметров>)` — то же самое, что `all`, но возвращаемый ассоциативный массив, наоборот, не включает параметры, перечисленные в *списке*. Можно либо перечислить строковые имена параметров непосредственно в вызове метода, либо оформить их как массив. Пример:

```
$data = $request->except(["desc", "order"]);
```

- `has(<имя параметра>)` — возвращает `true`, если параметр с указанным именем присутствует в запросе и в качестве значения имеет непустую строку, и `false` в противном случае;
- `exists(<имя параметра>)` — возвращает `true`, если параметр с указанным именем присутствует в запросе, даже если в качестве значения имеет пустую строку, и `false` в противном случае.

Для получения значения GET- или POST-параметра с указанным именем также можно использовать упомянутую ранее функцию `request`:

```
request(<имя параметра>[, <значение по умолчанию>])
```

Примеры:

```
$name = request("name");
$order = request("order", 0);
```

Работа с базой данных

Получив от посетителя запрос с интернет-адресом и выделив из интернет-адреса параметр с номером записи, контроллер будет искать эту запись в таблице базы данных. Кроме того, ему может понадобиться извлечь из базы целый набор записей, скорее всего, отфильтрованных и отсортированных согласно каким-то критериям.

Для работы с базой данных фреймворк Laravel предоставляет целый ряд мощных инструментов.

Простая выборка записей

Проще всего выбрать запись по ее номеру, извлечь значения ее полей и получить связанные с ней записи. Для этого понадобится написать, возможно, всего лишь одно выражение.

Поиск записей по их номерам

Проще всего найти запись по ее номеру, то есть по значению ключевого поля. Для этого достаточно вызвать всего один метод из описанных далее.

Метод `find` выполняет поиск записи по ее номеру, также есть возможность найти сразу несколько записей по их номерам:

```
<класс модели>::find(<номер>|<список номеров>[, <список полей>])
```

В качестве первого аргумента можно указать как числовой номер искомой записи, так и массив с номерами сразу нескольких записей. Вторым аргументом можно указать массив с именами полей, которые следует извлечь, — если он не указан, будут извлечены все поля записи.

Если первым аргументом был указан номер искомой записи, в качестве результата будет возвращен объект модели, представляющий найденную запись, или `null` в случае безуспешного поиска.

Если же первым аргументом указать массив номеров, будет возвращена коллекция, каждый элемент которой является объектом класса модели, представляющим одну из найденных записей. Эта коллекция будет пуста, если ни одной записи найти не удалось.

Вот пара примеров:

- ищем категорию с номером 1:

  ```
  $cat = Category::find(1);
  ```

- ищем категории с номерами 1, 2 и 3, после чего перебираем элементы полученной коллекции — найденные записи и выполняем их обработку:

  ```
  $cats = Category::find([1, 2, 3]);
  foreach ($cats as $cat) {
    // Обрабатываем полученные записи
  }
  ```

Аналогичный метод `findOrFail` в случае неуспешного поиска генерирует страницу с сообщением об отсутствии запрашиваемой страницы (ошибка 404).

Еще один полезный метод:

```
findMany(<список номеров>[, <список полей>])
```

работает так же, как метод `find` при указании в качестве первого аргумента массива с номерами записей, но выполняется несколько быстрее.

Выборка всех записей

Выбрать все записи, хранящиеся в таблице, также несложно. Для этого следует вызвать метод `get` или `all`:

<класс модели>`::get|all([`*<список полей>*`])`

Если массив со списком выбираемых из записей полей не указан, будут выбраны все поля. Метод возвращает коллекцию выбранных записей.

Вот пара примеров:

- выбираем все категории:

```
$cats = Category::get();
```

- выбираем все категории, извлекая из них только поля `name` и `description`:

```
$cats = Category::get(["name", "description"]);
```

Выборка первой записи

Извлечь самую первую запись таблицы можно, вызвав метод `first`:

<класс модели>`::first([`*<список полей>*`])`

Он вызывается так же, как метод `get`, но возвращает объект модели, представляющий первую запись, или `null`, если таблица не содержит ни одной записи.

В следующем примере мы извлекаем первую категорию в таблице:

```
$firstCat = Category::first();
```

Аналогичный метод `firstOrFail` вызывается и работает так же, но, если таблица не содержит записей, генерирует страницу с сообщением об отсутствии запрашиваемой страницы (ошибка 404).

Получение значений полей записи

Получить значения, хранящиеся в полях найденной записи, очень просто. Для этого достаточно воспользоваться свойствами класса модели, чьи имена совпадают с именами соответствующих им полей таблицы.

В следующем примере мы извлекаем категорию № 1 и получаем ее название и описание:

```
$cat = Category::find(1);
$name = $cat->name;
$desc = $cat->description;
```

Получение связанных записей

Из объекта модели, представляющей найденную запись, мы можем без проблем извлечь связанные с ней записи, также представленные объектами соответствующих моделей.

Класс первичной модели поддерживает свойство, чье имя совпадает с именем связи. (Об именах связей и вообще задании связей было рассказано в *главе 28*.) Это свойство хранит:

- коллекцию записей вторичной модели, связанных с записью первичной модели, если установлена связь «один-ко-многим» или «многие-ко-многим»;
- единственную связанную запись вторичной модели, если установлена связь «один-к-одному».

В следующем примере мы находим категорию № 1 и извлекаем из ее свойства `subcategories` (чье имя совпадает с именем метода, устанавливающего связь, — `subcategories`) коллекцию связанных подкатегорий:

```
$cat = Category::find(1);
$subcats = $cat->subcategories;
```

В классе вторичной модели также присутствует свойство, имя которого совпадает с именем метода, что объявлен в этом же классе и устанавливает связь. Это свойство хранит:

- единственную запись вторичной модели, связанную с записью первичной модели, если установлена связь «один-ко-многим» или «один-к-одному»;
- коллекцию связанных записей вторичной модели, если установлена связь «многие-ко-многим».

В следующем примере мы находим подкатегорию № 2, извлекаем из ее свойства `cat` (чье имя совпадает с именем метода, устанавливающего связь, — `cat`) связанную категорию, после чего получаем название последней:

```
$subcat = Subcategory::find(2);
$cat = $subcat->category;
$catName = $cat->name;
```

> ***Внимание!***
>
> Если для получения записи в вызове метода был указан список извлекаемых из записи полей, и поле, участвующее в образовании связи, в этом списке отсутствует, при попытке получить связанные записи возникнет ошибка. Например, этот код вызовет ошибку:
>
> ```
> $subcat = Subcategory::find(2, ["name", "order"]);
> $cat = $subcat->cat;
> $catName = $cat->name;
> ```
>
> поскольку в списке извлекаемых полей, указанном в вызове метода `find`, отсутствует поле, которое участвует в установлении связи. Чтобы этот код выполнился, нам следует изменить выражение вызова метода `find` на следующее:
>
> ```
> $subcat = Subcategory::find(2, ["name", "order", "category_id"]);
> ```
>
> Здесь предполагается, что упомянутое ранее поле называется `category_id`.

Создание запросов к базе данных

Часто бывает необходимо извлечь из таблицы базы данных целый набор записей, возможно, отфильтрованных и отсортированных согласно какому-либо критерию, или, говоря другими словами, создать *запрос к базе данных*. Для этого мы также можем воспользоваться инструментами, предоставляемыми моделями и описанными далее.

Эти инструменты — суть методы класса `Builder`, представляющего создаваемый запрос к базе данных. Каждый из них возвращает в качестве результата тот же объект того класса, у которого был вызван, вследствие чего мы можем просто сцеплять вызовы этого метода, записывая их друг за другом. Однако самый первый метод должен быть вызван у класса модели таким образом, как вызывается статический метод.

Для извлечения набора отфильтрованных и отсортированных записей в этом случае применяются рассмотренные ранее методы `get` или `first`.

Фильтрация записей

Фильтрация записей на основе какого-либо критерия при их выборке — вероятно, самая часто выполняемая операция. Осуществить фильтрацию записей можно, применив описанные далее методы.

Метод `where` указывает выбрать только те записи, в которых поле с заданным именем хранит указанное значение. Этот метод имеет три формата вызова:

- `where(<имя поля>[, <оператор сравнения>], <значение>[, <логический оператор>])`

  Имя поля указывается в виде строки. В качестве оператора сравнения могут быть заданы: `"="` (равно), `"<"`, `"<="`, `">"`, `">="`, `"<>"` (не равно) и `"like"` (подобно). Если оператор не указан, будет выполнена операция сравнения с применением оператора `"="`.

  В случае использования оператора `"like"` в *значении* могут применяться литералы `%` (совпадает с произвольным количеством любых символов), `_` (совпадает с одним любым символом), `\%` (символ процента) и `_` (символ подчеркивания).

  Логический оператор имеет смысл указывать, если до этого уже был выполнен вызов метода `where`, задающего какие-то критерии фильтрации, и к ним следует добавить критерии, задаваемые текущим вызовом этого метода, с применением какого-либо логического оператора: И или ИЛИ. Он указывается в виде строки `"and"` (логическое И) или `"or"` (логическое ИЛИ). Если логический оператор не указан, для связывания заданных условий будет применен оператор логическое И.

  Вот несколько примеров:

  - ищем категорию, поле `id` которой хранит значение 1:

    ```
    $cats = Category::where("id", 1)->first();
    ```

- ищем категории, поле `order` которых хранит значения больше 2:

  ```
  $cats = Category::where("order", ">", 2)->get();
  ```

- ищем категории, у которых значение поля `name` начинается с символов `"Web"` И поле `order` хранит значение больше нуля:

  ```
  $cats = Category::where("name", "like", "Web%")
  ->where("order", ">", 0)->get();
  ```

- ищем категории, у которых значение поля `name` начинается с символов `"Web"` ИЛИ поле `order` хранит значение большее нуля:

  ```
  $cats = Category::where("name", "like", "Web%")
  ->where("order", ">", 0, "or")->get();
  ```

❒ `where(`*`<массив условий>`*`)`

Массив условий должен содержать вложенные массивы, имеющие следующие элементы: имя поля, оператор сравнения, значение и логический оператор. Из них обязательными являются только имя поля и значение. Указываются они в том же виде, что и в случае первого формата вызова метода `where`.

Вот пара примеров:

- ищем категории, у которых значение поля `name` начинается с символов `"Web"` И поле `order` хранит значение большее нуля:

  ```
  $cats = Category::where([["name", "like", "Web%"],
  ["order", ">", 0]])->get();
  ```

- ищем категории, у которых значение поля `name` начинается с символов `"Web"` ИЛИ поле `order` хранит значение большее нуля:

  ```
  $cats = Category::where([["name", "like", "Web%"],
  ["order", ">", 0, "or"]])->get();
  ```

❒ `where(`*`<функция, задающая условия>`*`)`

Функция, задающая условия, должна принимать в качестве параметра объект класса `Builder`, представляющий создаваемый запрос. В своем теле она должна устанавливать критерии фильтрации, которые при генерировании SQL-запроса будут взяты в круглые скобки.

В следующем примере мы извлекаем категории, у которых:

- в поле `order` хранится значение больше нуля;
- И поле `desc` содержит строку `"Python"` ИЛИ `"Ruby"`:

  ```
  $cats = Category::where("order", ">", 0)
  ->where(function ($query) {
    $query->where([["desc", "like", "%Python%"],
    ["desc", "like", "%Ruby%", "or"]]);
  })->get();
  ```

Метод `orWhere` аналогичен вызову метода `where`, у которого четвертым аргументом указан оператор логического ИЛИ (`"or"`). Четвертый аргумент у этого метода не указывается.

Также поддерживаются описанные далее методы, задающие более сложные условия. Эти методы также поддерживают указание оператора сравнения и логического оператора, которые в этом случае задаются в том же формате, что и у метода `where`.

- `whereBetween(`*<имя поля>*`, `*<минимум и максимум>*`[, `*<логический оператор>*`][, `*<логическое НЕ?>*`])` — ищет записи, где в поле с указанным *именем* хранится значение, укладывающееся в диапазон между заданными *минимумом и максимумом*, если четвертым аргументом указано `false` или если четвертый аргумент вообще отсутствует, или значение, не укладывающееся в этот диапазон, если четвертым аргументом передано `true`. Минимум и максимум задаются в виде чисел, являющихся двумя элементами массива, который передается вторым аргументом.

  В следующем примере мы извлекаем категории с номерами от 1 до 5:

  ```
  $cats = Category::whereBetween("id", [1, 5])->get();
  ```

- `whereNotBetween(`*<имя поля>*`, `*<минимум и максимум>*`[, `*<логический оператор>*`])` — ищет записи, где в поле с указанным *именем* хранится значение, находящееся вне диапазона между заданными *минимумом и максимумом*. Минимум и максимум задаются в виде чисел, являющихся двумя элементами массива, который передается вторым аргументом. Аналогичен вызову метода `whereBetween` с передачей в качестве четвертого аргумента значения `true`.

  В следующем примере мы извлекаем все категории, за исключением тех, чьи номера находятся в диапазоне от 100 до 150:

  ```
  $cats = Category::whereNotBetween("id", [100, 150])->get();
  ```

- `whereIn(`*<имя поля>*`, `*<массив значений>*`[, `*<логический оператор>*`][, `*<логическое НЕ?>*`])` — ищет записи, где в поле с указанным *именем* хранится значение, совпадающее со значением одного из элементов *массива*, если четвертым аргументом указано `false` или если четвертый аргумент вообще отсутствует, или значение, не совпадающее ни с одним из этих элементов, если четвертым аргументом передано `true`.

  В следующем примере мы ищем подкатегории, относящиеся к категориям с номерами 1, 3 и 8:

  ```
  $subcats = Subcategory::whereIn("category", [1, 3, 8])->get();
  ```

- `whereNotIn(`*<имя поля>*`, `*<массив значений>*`[, `*<логический оператор>*`])` — ищет записи, где в поле с указанным *именем* хранится значение, не совпадающее ни с одним из элементов *массива*. Аналогичен вызову метода `whereIn` с передачей в качестве четвертого аргумента значения `true`.

- `whereNull(`*<имя поля>*`[, `*<логический оператор>*`][, `*<логическое НЕ?>*`])` — ищет записи, в которых поле с указанным *именем* хранит `null`, если третьим аргументом указано `false` или если третий аргумент вообще отсутствует, или значение, отличное от `null`, если третьим аргументом передано `true`.

В следующем примере мы ищем категории, в которых поле `description` (описание) не заполнено:

```
$cats = Category::whereNull("description")->get();
```

- `whereNotNull(<имя поля>[, <логический оператор>])` — ищет записи, в которых поле с указанным *именем* хранит значение, отличное от `null`. Аналогичен вызову метода `whereNull` с передачей в качестве третьего аргумента значения `true`.
- `whereDate(<имя поля>[, <оператор сравнения>], <дата>[, <логический оператор>])` — сравнивает значение *поля* с указанным именем и заданное значение *даты* и извлекает только те записи, для которых выполняется это сравнение. Дата задается в виде строки в формате *<год>-<номер месяца>-<число>*. Отметим, что в этом случае месяцы имеют номера от 1 до 12.

  В следующем примере мы ищем статьи, созданные в декабре 2016 года:

  ```
  $articles = Article::whereDate("created_at", ">=", "2016-12-01")
  ->whereDate("created_at", "<=", "2016-12-31")->get();
  ```

- `whereDay(<имя поля>[, <оператор сравнения>], <число>[, <логический оператор>])` — сравнивает значение *поля* с указанным именем (оно должно хранить дату или дату и время) и заданное *число* и извлекает только те записи, для которых выполняется это сравнение.
- `whereMonth(<имя поля>[, <оператор сравнения>], <номер месяца>[, <логический оператор>])` — сравнивает значение *поля* с указанным именем (оно должно хранить дату или дату и время) и заданный *месяц* (указывается в виде числа от 1 до 12) и извлекает только те записи, для которых выполняется это сравнение.

  В следующем примере мы ищем статьи, исправленные в ноябре или декабре:

  ```
  $articles = Article::whereMonth("updated_at", ">=", "11")->get();
  ```

- `whereYear(<имя поля>[, <оператор сравнения>], <год>[, <логический оператор>])` — сравнивает значение *поля* с указанным именем (оно должно хранить дату или дату и время) и заданный *год* и извлекает только те записи, для которых выполняется это сравнение.
- `whereColumn(<имя поля 1>[, <оператор сравнения>], <имя поля 2>[, <логический оператор>] | <массив условий>)` — сравнивает значения *поля 1* и *поля 2* и извлекает только те записи, для которых выполняется это сравнение. Здесь также может быть указан *массив условий*, который должен быть записан в том же формате, как и в случае метода `where`.

  Вот пара примеров:

  - ищем статьи, у которых поля `created_at` и `updated_at` хранят одинаковые значения, то есть статьи, не изменявшиеся после публикации:

    ```
    $articles = Article::whereColumn("created_at", "updated_at")
    ->get();
    ```

  - ищем статьи, у которых значение поля `created_at` меньше, чем у поля `updated_at`, то есть статьи, изменявшиеся после публикации:

```
$articles = Article::whereColumn("created_at", "<", "updated_at")
->get();
```

Методы `orWhereBetween`, `orWhereNotBetween`, `orWhereIn`, `orWhereNotIn`, `orWhereNull`, `orWhereNotNull`, `orWhereDate` и `orWhereColumn` выполняют те же задачи, что и аналогичные методы (`whereBetween`, `whereIn` и т. д.), в случае задания логического оператора ИЛИ (`"or"`). Аргумент логического оператора у таких методов не указывается.

Фильтрация по наличию или отсутствию связанных записей

Иногда бывает нужно извлечь лишь те записи, у которых имеются или, напротив, отсутствуют связанные записи. Для этого нужно применить один из описанных далее методов.

- `has(`*<имя связи>*`[, `*<оператор сравнения>*`][, `*<количество связанных записей>*`][, `*<логический оператор>*`][, `*<условие>*`])` — выполняет сравнение количества записей, доступных через связь с указанным *именем*, с заданным *количеством* при помощи заданного *оператора сравнения* и извлекает только записи текущей таблицы, для которых выполняется это сравнение. Если *оператор сравнения* не указан, выполняется сравнение с применением оператора >= (больше или равно); если не указано *количество записей*, оно принимается равным 1.

  Вот пара примеров:

  - извлекаем только те подкатегории, к которым относится хотя бы одна статья:

    ```
    $subcats = Subcategory::has("articles")->get();
    ```

  - извлекаем только те подкатегории, к которым относятся больше пяти статей:

    ```
    $subcats = Subcategory::has("articles", ">", 5)->get();
    ```

  Если нужно подсчитывать не все связанные записи, а только удовлетворяющие определенному критерию, мы укажем это *условие* в виде функции. Она должна принимать один параметр — объект класса `Builder`, представляющий запрос, с помощью которого в ее теле будет формироваться критерий фильтрации связанных записей.

  В следующем примере мы извлекаем только те подкатегории, к которым относятся больше пяти статей, опубликованных в 2016 году:

  ```
  $subcats = Subcategory::has("articles", ">", 5, function ($query) {
    $query->whereYear("created_at", 2016);
  })->get();
  ```

- `doesntHave(`*<имя связи>*`[, `*<логический оператор>*`][, `*<условие>*`])` — извлекает только те записи текущей таблицы, что не имеют записей, доступных через связь с указанным *именем*. Также возможно указание дополнительного *условия* в виде функции (см. ранее).

  В следующем примере мы получаем первую категорию, к которой не относится ни одной подкатегории:

  ```
  $cats = Category::doesntHave("subcategories")->first();
  ```

- `whereHas(`*<имя связи>*`, `*<условие>*`[, `*<оператор сравнения>*`][, `*<количество связанных записей>*`])` — упрощенный вариант метода `has`, предназначенный для указания дополнительного *условия* в виде функции. Заданный ими критерий фильтрации связывается с указанными ранее с применением оператора логическое И.
- `whereDoesntHave(`*<имя связи>*`, `*<условие>*`)` — упрощенный вариант метода `doesntHave`, предназначенный для указания дополнительного *условия* в виде функции. Заданный ими критерий фильтрации связывается с указанными ранее с применением оператора логическое И.

  В следующем примере мы выбираем только те подкатегории, к которым относится более пяти статей, опубликованных в 2016 году, И ни одной статьи, исправленной в январе:

```
$subcats = Subcategory::whereHas("articles", function ($query) {
  $query->whereYear("created_at", 2016);
}, ">", 5)->whereDoesntHave("articles", function ($query) {
  $query->whereMonth("updated_at", 1);
})->get();
```

- `orHas(`*<имя связи>*`[, `*<оператор сравнения>*`][, `*<количество связанных записей>*`])` — вариант метода `has`, связывающий заданный в нем критерий фильтрации с указанными ранее с применением оператора логическое ИЛИ. Не позволяет указать условие в виде функции.
- `orWhereHas(`*<имя связи>*`, `*<условие>*`[, `*<оператор сравнения>*`][, `*<количество связанных записей>*`])` — вариант метода `whereHas`, связывающий заданный в нем критерий фильтрации с указанными ранее с применением оператора логическое ИЛИ.

  В следующем примере мы выбираем только те подкатегории, к которым относится более пяти статей, опубликованных в 2016 году, ИЛИ хотя бы одна статья, исправленная в январе:

```
$subcats = Subcategory::whereHas("articles", function ($query) {
  $query->whereYear("created_at", 2016);
}, ">", 5)->orWhereHas("articles", function ($query) {
  $query->whereMonth("updated_at", 1);
})->get();
```

Сортировка записей

Отсортировать записи средствами Laravel еще проще, чем отфильтровать. Для этого предусмотрено всего четыре метода с очень простыми форматами вызова.

- `orderBy(`*<имя поля>*`[, `*<направление сортировки>*`])` — сортирует записи по значению поля с указанным *именем*. Если вторым аргументом передана строка `"asc"` или если второй аргумент вообще не указан, будет выполнена сортировка по возрастанию значений поля, если же задать строку `"desc"` — то по убыванию.

Вот пара примеров:

- сортируем категории по их названиям:

```
$cats = Category::orderBy("name")->get();
```

- сортируем категории сначала по названиям, а потом — по убыванию значения поля `order` (порядка следования категорий):

```
$cats = Category::orderBy("name")->orderBy("order", "desc")
->get();
```

- `latest([<имя поля>])` — сортирует записи по полю даты или даты и времени с указанным *именем* по убыванию. Если *имя поля* не указано, выполняет сортировку по полю `created_at`.

  В следующем примере мы сортируем статьи по убыванию даты их добавления:

```
$comments = Comment::latest()->get();
```

- `oldest([<имя поля>])` — сортирует записи по полю даты или даты и времени с указанным *именем* по возрастанию. Если *имя поля* не указано, выполняет сортировку по полю `created_at`.
- `inRandomOrder` — сортирует записи в случайном порядке. Не принимает аргументов.

Указание выбираемых полей

Методы `get`, `all` и `first` предоставляют нам возможность задать список полей, которые следует выбрать из таблицы. Еще для этого может применяться метод `select`, который предоставляет дополнительные возможности:

```
select(<список полей>)
```

Этот метод принимает произвольное количество аргументов, заданных в виде строк. Этими строками могут быть как имена полей, так и конструкции вида:

```
<имя поля> as <псевдоним поля>
```

если у поля нужно задать псевдоним.

Вот несколько примеров:

- получаем категории, для которых извлекаем из таблицы лишь имена и описания, отсортировав их по именам:

```
$cats = Category::select("name", "description")->orderBy("name")
->get();
```

- то же самое, только для поля описания (`description`) указываем псевдоним `desc`. Теперь поле описания будет доступно под этим псевдонимом:

```
$cats = Category::select("name", "description as desc")
->orderBy("name")->get();
```

  Получаем описание первой из извлеченных категорий через псевдоним `desc`:

```
$catDesc = $cats[0]->desc;
```

Метод `addSelect` добавляет к заданному ранее методом `select` набору выбираемых полей те, что указаны в его вызове. Его формат вызова схож с таковым у метода `select`.

Выборка уникальных записей

Не принимающий аргументов метод `distinct` указывает Laravel извлечь только уникальные записи:

```
$cats = Category::select("description")->distinct()->get();
```

Здесь мы получаем список уникальных описаний категорий.

Связывание таблиц

Laravel предлагает средства для связывания таблиц и для извлечения полей обеих таблиц — как первичной, так и вторичной. Для этого применяются три метода и особый формат записи имен выбираемых полей.

> ***На заметку***
>
> Если нужно всего лишь получить записи, связанные с текущей записью, имеет смысл использовать средства, описанные ранее, в *разд. «Получение связанных записей»*. Описанные здесь средства следует задействовать, если необходимо получить набор записей, каждая из которых, помимо собственных полей, включает также поля связанной записи. Впрочем, с подобного рода наборами иметь дело приходится довольно часто...

Саму связь устанавливает метод `join`:

```
join(<имя связываемой таблицы>, <имя поля текущей таблицы, участвующего в
образовании связи>, <имя поля связываемой таблицы>[, <тип связи>])
```

Все аргументы задаются в виде строк. *Тип связи* может быть следующим:

- `"inner"` — выбирает лишь связанные записи (поведение по умолчанию);
- `"left"` — выбирает все записи текущей таблицы и лишь связанные с ними записи в связываемой таблице;
- `"right"` — выбирает все записи в связываемой таблице и лишь связанные с ними записи в текущей таблице.

Если в обеих связываемых таблицах присутствуют поля с одинаковыми именами, нам придется явно указать, к какой таблице относится каждое такое поле, иначе MySQL не сможет выполнить запрос. Имена выбираемых полей задаются с применением уже знакомого нам метода `select`, а сами поля — в формате:

```
<имя таблицы>.<имя поля> as <псевдоним поля>
```

Псевдоним также следует указать — хотя бы для одного из полей с одинаковыми именами, чтобы впоследствии к такому полю можно было без проблем обратиться.

Если указать вместо *имени поля* символ `*`, будут извлечены все поля таблицы с указанным *именем*.

Вот пара примеров:

- связываем таблицы subcategories и categories, получаем названия (subcategories.name), описания (subcategories.description) подкатегорий и названия категорий, к которым они относятся (categories.name, доступны по псевдониму catname):

```
$subcats = Subcategory::select("subcategories.name",
"subcategories.description", "categories.name as catname")
->join("categories", "subcategories.category", "categories.id")
->get();
```

- то же самое, только извлекаем все поля таблицы subcategories:

```
$subcats = Subcategory::select("subcategories.*",
"categories.name as catname")
->join("categories", "subcategories.category", "categories.id")
->get();
```

Использование агрегатных функций. Группировка записей

Агрегатные функции, рассмотренные нами в *главе 25*, — замечательное средство для получения статистической информации о записях таблицы: их количества, среднего, вычисленного на основе значений какого-либо поля, и др. Laravel предоставляет нам целый набор инструментов для этого, равно как и для выполнения группировки записей (которая также рассматривалась в *главе 25*).

Получение количества связанных записей

Проще всего получить количество записей, связанных с текущей записью. Для этого применяется метод withCount:

```
withCount(<имя связи>|[<список имен связей, возможно, с критериями
фильтрации записей>])
```

В качестве аргумента можно указать:

- одно *имя связи* в виде строки — тогда будут подсчитаны все связанные записи, относящиеся к этой связи;
- *список имен связей, возможно, с критериями фильтрации записей* — тогда будут подсчитаны связанные записи, относящиеся ко всем указанным связям. Если для какой-то связи дополнительно задан критерий фильтрации, то будут подсчитаны все записи для этой связи, что удовлетворяют этому критерию.

  Список имен связей указывается в виде комбинированного массива:

  - если нужно подсчитать все записи, относящиеся к какой-либо связи, следует указать имя этой связи в виде обычного элемента массива;
  - если нужно указать критерий фильтрации, то следует задать в качестве ключа элемента имя связи, а в качестве значения — функцию, которая будет прини-

мать в качестве единственного параметра объект класса `Builder`, представляющий запрос, и в своем теле, собственно, задавать критерий фильтрации.

Этот метод создает в каждой модели свойство с именем вида *<имя связи>*`_count`, хранящее количество записей, доступных через связь с указанным в его вызове *именем*.

Вот пара примеров:

- извлекаем первую категорию и получаем количество относящихся к ней подкатегорий:

```
$cat = Category::withCount("subcategories")->first();
$subcatCount = $cat->subcategories_count;
```

- извлекаем первую подкатегорию и получаем количество относящихся к ней статей, исправленных в 2016 году:

```
$subcat = Subcategory::withCount(["articles" => function ($query) {
  $query->whereYear("updated_at", 2016);
}])->first();
$articlesCount = $subcat->articles_count;
```

Использование агрегатных функций применительно ко всем записям

Если нужно использовать какую-то агрегатную функцию применительно ко всем записям таблицы (возможно, после их фильтрации), следует вызвать один из описанных далее методов:

- `count` — возвращает количество записей. Не принимает аргументов;
- `min(`*<имя поля>*`)` — возвращает минимальное значение из хранящихся в поле с указанным *именем*;
- `max(`*<имя поля>*`)` — возвращает максимальное значение из хранящихся в поле с указанным *именем*;
- `sum(`*<имя поля>*`)` — возвращает сумму всех значений, хранящихся в поле с указанным *именем*;
- `avg(`*<имя поля>*`)` — возвращает арифметическое среднее, вычисленное на основе значений, что хранятся в поле с указанным *именем*;
- `average(`*<имя поля>*`)` — то же самое, что и `avg`.

Все эти функции возвращают обычное числовое значение.

В следующем примере мы получаем количество всех статей, созданных в 2016-м и последующих годах:

```
$newArticleCount = Article::whereYear("created_at", ">=", 2016)->count();
```

Использование агрегатных функций применительно к сгруппированным записям

Но гораздо чаще приходится использовать агрегатные функции применительно к группам записей. Для этого придется сначала сгруппировать эти записи по значению какого-либо поля или сразу нескольких полей.

Группировка записей задается вызовом метода `groupBy`:

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

Можно указать как *имя* одного поля, так и *массив* из нескольких имен полей. В последнем случае группировка будет выполнена по всем этим полям.

Агрегатная функция, которая должна быть вычислена на основе сформированных групп, указывается в вызове рассмотренного нами ранее метода `select`. Однако указать ее напрямую нельзя. Дело в том, что Laravel выполняет предварительную обработку всех значений, заданных в вызовах методов, которые рассматриваются в этой главе, и вызов агрегатной функции, будучи подвергнутым такой обработке, превратится в строку, которую MySQL не сможет разобрать. Итогом станет сообщение об ошибке в коде SQL-запроса.

Чтобы поместить в вызов метода `select` вызов агрегатной функции, следует применить метод `raw` фасада `DB`:

```
DB::raw(<строка с вызовом агрегатной функции>)
```

В следующем примере мы связываем таблицы `categories` и `subcategories`, указываем группировку по полю `subcategories.id`, извлекаем поле `categories.name` и количество записей таблицы `subcategories`, связанных с каждой записью таблицы `categories`, задав для него псевдоним `subcat_count`. В результате мы получим набор записей с полями `name` (название категории) и `subcat_count` (количество подкатегорий, относящихся к этой категории):

```
use Illuminate\Support\Facades\DB;
. . .
$cats = Category::select("categories.name",
DB::raw("count(subcategories.id) as subcat_count"))
->join("subcategories", "categories.id", "subcategories.category")
->groupBy("categories.id")->get();
```

Фильтрация и сортировка групп записей

Еще имеется возможность отфильтровать сформированные методом `groupBy` группы записей по какому-либо критерию. Для этого применяется методы, описанные далее.

- `having(<имя поля>[, <оператор сравнения>], <значение>[, <логический оператор>])` — задает критерий на основе значения поля. Аргументы аналогичны тем, что указываются при вызове метода `where` (см. ранее).

В следующем примере мы получаем имена категорий и количество связанных с ними подкатегорий только для тех категорий, чьи номера начинаются с четырех:

```
$cats = Category::select("categories.name",
DB::raw("count(subcategories.id) as subcat_count"))
->join("subcategories", "categories.id", "subcategories.category")
->groupBy("categories.id")->having("categories.id", ">", 3)->get();
```

- `orHaving(`*<имя поля>*`[, `*<оператор сравнения>*`], `*<значение>*`)` — аналогичен вызову метода `having` с указанием оператора логического ИЛИ (`"or"`).
- `havingRaw(`*<условие>*`[, `*<логический оператор>*`])` — позволяет задать критерий на основе результата, возвращаемого агрегатной функцией. Условие указывается в виде строки и должно представлять собой выражение SQL, сравнивающее возвращенный агрегатной функцией результат с нужным значением.

  В следующем примере мы получаем имена и количество связанных подкатегорий только для тех категорий, что содержат более пяти подкатегорий:

```
$cats = Category::select("categories.name",
DB::raw("count(subcategories.id) as subcat_count"))
->join("subcategories", "categories.id", "subcategories.category")
->groupBy("categories.id")->havingRaw("count(subcategories.id) > 5")
->get();
```

- `orHavingRaw(`*<условие>*`)` — аналогичен вызову метода `havingRaw` с указанием оператора логического ИЛИ (`"or"`).

Для сортировки групп записей можно использовать как уже знакомый нам метод `orderBy`, так и метод `orderByRaw`, с помощью которого можно задать критерии сортировки на основе результата, возвращенного агрегатной функцией:

`orderByRaw(`*<критерий сортировки>*`)`

В следующем примере мы получаем имена категорий и количество связанных с ними подкатегорий, отсортировав результат по убыванию количества подкатегорий:

```
$cats = Category::select("categories.name",
DB::raw("count(subcategories.id) as subcat_count"))
->join("subcategories", "categories.id", "subcategories.category")
->groupBy("categories.id")->orderByRaw("count(subcategories.id)", "desc")
->get();
```

Ограничение количества выбираемых записей

Мы также имеем возможность указать количество выбираемых записей и количество записей, которые следует пропустить перед собственно выборкой.

Для указания количества выбираемых записей применяются методы `limit` или `take`:

`limit|take(`*<количество выбираемых записей>*`)`

Количество выбираемых записей задается в виде числа.

Для указания количества записей, которые следует пропустить перед собственно выборкой, служат методы offset или skip:

offset|skip(<*количество пропускаемых записей*>)

Количество пропускаемых записей также задается в виде числа.

В следующем примере мы выбираем все статьи с 16-й по 20-ю:

```
$articles = Article::offset(15)->limit(5)->get();
```

Специальные случаи выборки записей

Помимо уже знакомых нам методов all, get и first, Laravel поддерживает набор методов для специальной выборки записей. Иногда они могут очень пригодиться.

- pluck(<*имя поля для выборки*>[, <*имя поля для формирования ключей*>]) — возвращает массив, содержащий все значения из поля с указанным в первом аргументе *именем*. Если указано *имя поля для формирования ключей*, будет возвращен ассоциативный массив, ключи элементов которого будут представлять собой значения, взятые из этого поля, а значения элементов — значения, взятые из поля, чье имя задано первым аргументом.

  Вот пара примеров:

  - извлекаем массив названий всех категорий и получаем название самой первой категории:

    ```
    $catNames = Category::pluck("name");
    $firstCatName = $catNames[0];
    ```

  - извлекаем ассоциативный массив, где в качестве ключей используются номера категорий (значения поля id), а значения — их названия, после чего получаем название категории № 2.

    ```
    $catNames = Category::pluck("name", "id");
    $firstCatName = $catNames["2"];
    ```

- implode(<*имя поля*>[, <*строка-разделитель*>]) — возвращает строку, составленную из всех значений, что хранятся в поле с указанным *именем*, разделенных *строкой-разделителем*. Если таковая не указана, будет использована пустая строка, и значения будут расположены вплотную друг к другу.

  В следующем примере мы получаем строку, составленную из названий всех категорий, разделенных запятыми:

  ```
  $catNames = Category::implode("name", ", ");
  ```

- exists — возвращает true, если запрос содержит хотя бы одну запись, и false в противном случае. Не принимает аргументов.

Использование пагинатора

На Web-страницах большие списки обычно выводятся постранично. Это позволяет сократить объем данных, загружаемых по сети, а также избавить посетителя от необходимости прокручивать огромный список в поисках нужной ему позиции.

Фреймворк Laravel предоставляет нам встроенный инструмент для разбиения списков записей на страницы — *пагинатор*. Даже два таких инструмента: упрощенный и полнофункциональный.

Упрощенный пагинатор

Упрощенный пагинатор позволяет вывести на страницу только гиперссылки вида **Вперед** и **Назад**, указывающие, соответственно, на следующую и предыдущую страницы. Он создается вызовом метода `simplePaginate`:

```
simplePaginate([<количество записей, выводящихся на странице>
[, <список извлекаемых полей>
[, <имя GET-параметра, хранящего номер выводимой страницы>
[, <номер страницы, выводящейся изначально>]]]])
```

В качестве результата этот метод возвращает набор записей, аналогичный генерируемому вызовом методов `all` или `get`, но содержащий только те записи, что относятся к текущей странице. Этот набор записей является объектом класса `Paginator`.

При создании пагинатора, как и в случае упомянутых методов, можно указать массив со *списком извлекаемых полей*.

Для передачи номера страницы, которая должна быть выведена, применяется особый GET-параметр. Пагинатор сам извлекает его из интернет-адреса — нам этого делать не придется.

Значения по умолчанию, используемые пагинатором, если не указаны соответствующие аргументы:

- количество записей, выводящихся на странице, — 15;
- список извлекаемых полей — все поля;
- имя GET-параметра, хранящего номер страницы, — `page`;
- номер страницы, выводящейся изначально, — 1.

Вот пара примеров:

- сортируем статьи по убыванию даты их последнего изменения и создаем на основе получившегося набора упрощенный пагинатор с параметрами по умолчанию:

  ```
  $pag = Article::latest("updated_at")->simplePaginate();
  ```

- то же самое, только указываем пагинатору выводить на каждой странице по 10 записей, извлекать из записей только поля `title` и `description`, использовать для передачи номера страницы GET-параметр `p` и изначально вывести вторую страницу:

  ```
  $pag = Article::latest("updated_at")->simplePaginate(10,
  ["title", "description"], "p", 2);
  ```

Класс `Paginator` поддерживает ряд методов, позволяющих получить различные сведения о текущей странице и самом пагинаторе. Эти методы приведены в табл. 30.1.

Таблица 30.1. *Методы класса* `Paginator`

Метод	**Описание**
`nextPageUrl()`	Возвращает строку с интернет-адресом, указывающим на следующую страницу пагинатора, или `null`, если это последняя страница
`previousPageUrl()`	Возвращает строку с интернет-адресом, указывающим на предыдущую страницу пагинатора, или `null`, если это первая страница
`url(`*<номер страницы>*`)`	Возвращает строку с интернет-адресом, указывающим на страницу с заданным *номером*
`currentPage()`	Возвращает номер текущей страницы
`hasMorePages()`	Возвращает `true`, если это не последняя страница с записями, и `false` в противном случае
`hasPages()`	Возвращает `true`, если это не единственная страница с записями, и `false` в противном случае
`count()`	Возвращает действительное количество записей, выведенных на текущей странице (может быть меньше количества, указанного при создании пагинатора, если это последняя страница)
`firstItem()`	Возвращает порядковый номер первой записи, выводящейся на текущей странице
`lastItem()`	Возвращает порядковый номер последней записи, выводящейся на текущей странице
`perPage()`	Возвращает количество записей, выводящееся на странице
`isEmpty()`	Возвращает `true`, если текущая страница пуста (не содержит записей), и `false` в противном случае

По умолчанию пагинатор генерирует интернет-адреса страниц на основе текущего интернет-адреса, что указывает на действие контроллера, в котором был создан сам пагинатор. Однако мы можем указать другой интернет-адрес, вызвав метод `setPath` класса `Paginator`:

```
<пагинатор>->setPath(<интернет-адрес>)
```

Также мы можем добавить к интернет-адресам, генерируемым пагинатором, набор произвольных GET-параметров и имя якоря. Для этого применяются методы `appends` и `fragment` класса `Paginator` соответственно:

```
<пагинатор>->appends(<имя параметра>, <значение параметра> |
[<список параметров>])
<пагинатор>->fragment(<имя якоря>)
```

В вызове метода `appends` можно указать либо *имя* и *значение* добавляемого GET-параметра в двух отдельных аргументах, либо один аргумент со *списком* добавляемых GET-параметров. Список добавляемых параметров должен представлять собой ассоциативный массив, где ключи элементов выступят именами GET-параметров, а значения элементов — значениями этих параметров.

Пример:

```
$pag->setPath("/articles/paged");
$pag->appends(["sort" => "updated_at", "search" => "html"]);
$pag->fragment("comments");
```

Полнофункциональный пагинатор

Полнофункциональный пагинатор используется в тех случаях, когда, помимо гиперссылок, указывающих на следующую и предыдущую страницы, нужно вывести набор гиперссылок, что ведут на отдельные страницы. Однако он обрабатывается дольше, чем упрощенный, и требует больше системных ресурсов.

Полнофункциональный пагинатор создается вызовом метода `paginate`, который вызывается так же, как метод `simplePaginate`, и представляется классом `LengthAwarePaginator`.

Единственное отличие этого класса от рассмотренного ранее класса `Paginator` — поддержка двух дополнительных методов:

- `lastPage` — возвращает номер последней страницы;
- `total` — возвращает общее количество страниц.

Оба этих метода не принимают аргументов.

Получение сведений о запросе

Запрос в Laravel представляется объектом класса `Request` из пространства имен `Illuminate\Http`. Этот класс может предоставить нам различную полезную информацию — от данных, введенных посетителем в Web-форму (как их получить, говорилось в начале этой главы), до текущего интернет-адреса.

Объект, представляющий запрос, можно получить через параметр метода-действия контроллера. Этот параметр должен стоять на первом месте, и для него следует указать в качестве типа класс `Request` (как показано в листинге 30.2).

Листинг 30.2

```
use Illuminate\Http\Request;
class CategoryController extends Controller {
  public function index(Request $request) {
    // Получаем сведения о запросе, хранящемся в параметре $request
  }
}
```

Класс `Request` поддерживает ряд полезных методов — в дополнение к тем, что были описаны в *разд. «Получение данных от посетителя»*. Эти методы приведены в табл. 30.2.

Таблица 30.2. *Дополнительные методы класса* `Request`

Метод	Описание
`fullUrl()`	Возвращает строку с интернет-адресом, включающим набор GET-параметров
`url()`	Возвращает строку с интернет-адресом без набора GET-параметров
`path()`	Возвращает строку с путем к запрошенной серверной программе, входящим в состав интернет-адреса
`is(`*<шаблон>*`)`	Возвращает `true`, если текущий интернет-адрес совпадает с заданным шаблоном, и `false` в противном случае. В шаблоне можно использовать литерал `*`, обозначающий любое количество символов
`method()`	Возвращает строку с обозначением метода, с применением которого были отправлены данные
`isMethod(`*<метод>*`)`	Возвращает `true`, если использовался указанный *метод* отправки данных, и `false` в противном случае
`root()`	Возвращает строку с интернет-адресом главной страницы сайта
`fullUrlWithQuery(`*<список GET-параметров>*`)`	Возвращает строку с текущим интернет-адресом, к которому добавлены GET-параметры, указанные в *списке*. Список должен представлять собой ассоциативный массив, где ключи элементов станут именами GET-параметров, а их значения — значениями этих параметров
`ajax()`	Возвращает `true`, если текущий запрос был выполнен с применением технологии AJAX, и `false` в противном случае
`secure()`	Возвращает `true`, если текущий запрос был выполнен по протоколу *HTTPS* (HTTP Secure, защищенная разновидность протокола HTTP), и `false` в противном случае
`ip()`	Возвращает строку с IP-адресом, с которого пришел текущий запрос
`header(`*<ключ>*`)`	Возвращает строку со значением параметра с указанным *ключом*, который был получен в составе заголовка запроса (обычно таким образом передаются всяческого рода служебные данные)

Вот несколько примеров:

- в теле метода-действия `index` получаем полный интернет-адрес, пришедший в составе запроса:

```
public function index(Request $request) {
  $url = $request->fullUrl();
  . . .
}
```

- проверяем, совпадает ли интернет-адрес с шаблоном `"categories/*"`, и, если так, выполняем какие-либо действия:

```
if ($request->is("categories/*")) {
  . . .
}
```

- получаем текущий интернет-адрес с добавленным GET-параметром `search`, для которого задано значение `"html"`:

```
$url = $request->fillUrlWithQuery(["search" => "html"]);
```

- получаем строку со значением параметра `User-Agent`, отправленного в составе заголовка запроса. В этом параметре передаются сведения о Web-обозревателе, отправившем запрос:

```
$ud = $request->header("User-Agent");
```

Получение путей к папкам фреймворка

Laravel поддерживает несколько функций, позволяющих получить полные пути к различным папкам фреймворка. Некоторые из этих функций, возвращающих пути к самым важным папкам, приведены в табл. 30.3.

Таблица 30.3. *Некоторые функции, возвращающие пути к папкам фреймворка*

Функция	Описание
`base_path`	Возвращает строку с полным путем папки проекта
`public_path`	Возвращает строку с полным путем папки *public*
`storage_path`	Возвращает строку с полным путем папки *storage*

Все эти функции могут принимать аргумент — строку с путем к папке или файлу. Этот путь будет добавлен к пути соответствующей папки, после чего возвращен в качестве результата.

В следующем примере мы формируем в переменной `file` полный путь к файлу file.zip, хранящемуся в папке app\public папки storage:

```
public function file() {
  $filePath = storage_path("/app/public/file.zip");
  . . .
}
```

Встроенная в PHP константа `DIRECTORY_SEPARATOR` хранит символ, использующийся в качестве разделителя имен папок и файлов в пути.

Вывод данных

Итак, данные получены, обработаны, и на их основе выработан какой-то результат. Как вывести его на экран?

Вывод посредством шаблона

Чтобы сгенерировать на основе полученного результата Web-страницу, использовав для этого заранее созданный шаблон (о шаблонах речь пойдет в *главе 31*), следует вызвать функцию `view`:

```
view(<имя шаблона>[, <контекст данных шаблона>)
```

Имя шаблона указывается по следующим правилам:

- указывается только та часть имени, что предшествует расширению blade.php;
- если файл с шаблоном находится непосредственно в папке resources\views, указывается только имя этого файла;
- если файл с шаблоном вложен в папку, вложенную в папку resources\views, путь к файлу указывается относительно этой папки, а имена папок, входящих в состав пути, разделяются символами точки (не слеша!). Также точкой разделяются имя последней папки и имя файла шаблона.

Контекст данных шаблона — это данные, которые будут выведены с применением шаблона, или, как говорят разработчики сайтов, *переданы шаблону для вывода*. Они задаются в виде ассоциативного массива, в котором ключи элементов станут именами переменных, из которых в шаблоне можно будет извлечь данные, а значения элементов — самими этими данными.

Если контекст данных не указан, шаблону не будет передано никаких данных (точнее, будет передан пустой массив).

Результат, возвращенный функцией `view`, — объект класса `View`, представляющий шаблон, — следует вернуть из метода-действия в качестве результата. Если этого не сделать, страница не будет отправлена.

Вот пара примеров:

- здесь мы в теле метода-действия `index` извлекаем все категории и выводим их на экран с помощью шаблона catindex.blade.php, который хранится непосредственно в папке resources\views. Отметим, что в коде шаблона набор категорий будет доступен через переменную `cats`, поскольку именно такой ключ мы указали в ассоциативном массиве, задающем контекст данных шаблона:

  ```
  public function index() {
    $cats = Categories::all();
    return view("catindex", ["cats" => $cats]);
  }
  ```

- то же самое, но здесь мы обращаемся к шаблону index.blade.php, хранящемуся в папке categories, что вложена в папку resources\views. Обратите внимание, как в этом случае задается имя шаблона:

  ```
  return view("categories.index", ["cats" => $cats]);
  ```

Метод `with` класса `View` позволяет добавить к контексту данных шаблона новые значения уже после вызова функции `view`:

```
<ответ>->with(<имя значения>, <значение> | <список значений>)
```

Здесь можно указать либо `список значений`, которые следует добавить, в виде ассоциативного массива такого же формата, что задается в вызове функции `view`, либо `имя значения` и само `значение` в виде отдельных аргументов.

Метод `with` в качестве результата возвращает тот же объект класса `View`, у которого был вызван.

В следующем примере мы выполняем вывод данных с применением шаблона index.blade.php без указания контекста данных, после чего добавляем в «пустой» контекст список категорий под именем `cats`, номер текущей категории 1 под именем `currentCatID` и заголовок сайта `"Публикации"` под именем `siteHeader`:

```
return view("index")->with("cats", $cats)
->with(["currentCatID" => 1, "siteHeader" => "Публикации"]);
```

Также нам может пригодиться метод `exists` фасада `View`, позволяющий проверить, существует ли шаблон с указанным именем:

```
View::exists(<имя шаблона>)
```

Метод возвращает `true`, если такой шаблон существует, и `false` в противном случае. Пример:

```
use Illuminate\Support\Facades\View;
. . .
if (View::exists("categories.edit")) {
  // Шаблон categories.edit существует
}
```

Вывод в формате JSON

Вероятно, выполнить вывод данных в формате JSON (что пригодится при реализации загрузки данных с применением AJAX) в Laravel проще, чем в прочих фреймворках. Это выполняется в три шага.

1. Вызывается функция `response`, которая возвращает объект класса `Response`, представляющий отправляемый Web-обозревателю ответ, и которой в этом случае не передается никаких аргументов.
2. У полученного на *шаге 1* объекта класса `Response` вызывается метод `json`:

   ```
   <ответ>->json(<данные>)
   ```

   `Данные` должны представлять собой ассоциативный массив, который и будет преобразован в формат JSON.
3. Возвращенный методом `json` результат следует вернуть из метода-действия в качестве результата.

Пример:

```
public function index() {
  $cats = Categories::all();
  return response()->json(["data" => $cats, "status" => 1]);
}
```

Вывод пагинатора в формате JSON

Вывести набор записей, разбитых на страницы с применением пагинатора, в формате JSON также несложно. Для этого достаточно вернуть пагинатор из действия контроллера в качестве результата. Пример:

```
public function index() {
  return Article::latest()->paginate();
}
```

Экземпляр объекта, в который будут преобразованы данные JSON, описывающие пагинатор, будет включать следующие свойства:

- `data` — собственно массив записей, выводящихся на текущей странице;
- `next_page_url` — интернет-адрес следующей страницы или `null`, если это последняя страница;
- `prev_page_url` — интернет-адрес предыдущей страницы или `null`, если это последняя страница;
- `current_page` — номер текущей страницы;
- `from` — порядковый номер первой записи, выводящейся на текущей странице;
- `to` — порядковый номер последней записи, выводящейся на текущей странице;
- `per_page` — количество записей, выводящееся на странице;
- `last_page` — номер последней страницы (поддерживается только полнофункциональным пагинатором);
- `total` — общее количество страниц (поддерживается только полнофункциональным пагинатором).

Отправка файлов

И, наконец, мы можем выполнить отправку файла с указанным путем, причем двумя способами.

В обоих случаях сначала следует вызвать описанную ранее функцию `response` без аргументов. Она вернет объект класса `Response`, представляющий отправляемый Web-обозревателю ответ.

Первый способ пригодится, если отправляемый файл должен быть загружен и сохранен на локальном диске. Реализуется он вызовом метода `download` класса `Response`:

```
<ответ>->download(<путь отправляемого файла>
[, <имя, под которым отправляемый файл будет сохранен на локальном
диске>])
```

Если *имя, под которым отправляемый файл будет сохранен на локальном диске*, не указано, файл будет сохранен под именем, указанным в *пути*.

Результат, возвращенный методом `download`, следует вернуть из метода-действия в качестве результата.

В следующем примере мы отправляем файл file.zip, хранящийся в папке app\public папки storage, Web-обозревателю под именем archive-file.zip:

```
return response()->download(storage_path("/app/public/file.zip"),
"archive-file.zip");
```

Второй способ позволит нам отправить Web-обозревателю файл не для загрузки, а для вывода на экран, — это может понадобиться для графических, аудио- и видеофайлов, статических страниц и т. п. Эту задачу выполняет метод `file` класса `Response`:

```
<ответ>->file(<путь отправляемого файла>)
```

В следующем примере мы отправляем для вывода на экран файл image.png, хранящийся в папке app\images папки storage:

```
return response()->file(storage_path("/app/images/image.png"));
```

Перенаправление

Перенаправление на другую страницу выполняется в динамических сайтах очень часто. Например, после сохранения вновь созданной статьи имеет смысл перенаправить посетителя на страницу со списком статей.

Перенаправление можно выполнить на обычный интернет-адрес, на предыдущую страницу, на именованный маршрут (об именованных маршрутах рассказывалось в *главе 29*), на маршрут с именем `home` и на действие контроллера.

- Перенаправление на обычный интернет-адрес выполняется вызовом функции `redirect` и возвратом в качестве результата (им станет объект класса `RedirectResponse`, представляющий перенаправление) из метода-действия результата, который она вернет:

  ```
  redirect(<целевой интернет-адрес>)
  ```

  Пример:

  ```
  return redirect("/categories");
  ```

- Перенаправление на предыдущую страницу выполняется вызовом не принимающей аргументов функции `back` и возвратом в качестве результата из метода-действия результата, который она вернет (это объект того же класса `RedirectResponse`). Пример:

  ```
  return back();
  ```

- Перенаправление на именованный маршрут выполняется вызовом функции `redirect` без указания аргументов и вызовом у возвращенного ей объекта класса `Redirector` метода `route`:

  ```
  <перенаправление>->route(<имя маршрута>[, <параметры>])
  ```

  Параметры будут подставлены в сформированный на основе маршрута с указанным *именем* интернет-адрес. Если же в маршруте отсутствует параметр с за-

данным именем, он будет добавлен к интернет-адресу в качестве GET-параметра.

`Параметры` указываются в виде комбинированного массива. Ключ элемента станет именем параметра, а значение элемента — значением этого параметра. Если в маршруте задействовано внедрение модели (за подробностями — к *главе 29*), соответствующий объект модели можно указать в качестве обычного элемента массива, без ключа.

Результат, возвращенный методом `route` (это объект класса `RedirectResponse`), следует вернуть из метода-действия в качестве результата.

Вот пара примеров:

- выполняем перенаправление на маршрут `articles.edit`, указав в качестве значения параметра `article` номер статьи:

  ```
  return redirect()->route("articles.edit",
  ["article" => $article->id]);
  ```

- выполняем перенаправление на маршрут `articles.edit`, в котором задействовано внедрение модели. Здесь мы указываем в качестве параметра саму статью:

  ```
  return redirect()->route("articles.edit", [$article]);
  ```

- Перенаправление на маршрут с именем `home` (обычно это главная страница сайта) производится вызовом метода `home` фасада `Redirect` и возвратом в качестве результата из метода-действия результата, который вернет этот метод. Пример:

  ```
  use Illuminate\Support\Facades\Redirect;
  . . .
  return Redirect::home();
  ```

- Перенаправление на действие контроллера производится вызовом функции `redirect` без указания аргументом и вызовом у возвращенного ей объекта класса `Redirector` метода `action`:

  ```
  <перенаправление>->action(<действие>[, <параметры>])
  ```

  `Действие` указывается в формате `<имя контроллера>@<имя действия>`. Параметры задаются в том же формате, что и в случае метода `route`. Пример:

  ```
  return redirect()->action("ArticleController@edit",
  ["article" => $article->id]);
  ```

Перенаправление с выводом всплывающих сообщений

Всплывающее сообщение в терминологии Laravel — это строковое значение, которое сохраняется на стороне сервера, существует только, пока генерируется следующая страница, может быть выведено на нее и удаляется после того, как следующая страница будет сгенерирована и отправлена.

Если необходимо, помимо выполнения перенаправления, вывести на целевой странице всплывающее сообщение, нужно перед возвратом из метода-действия результата, возвращенного одним из описанных ранее методов, вызвать у этого результата — объекта класса `RedirectResponse` — метод `with`:

```
<перенаправление>->with(<имя сообщения>, <содержимое сообщения> |
<список сообщений>)
```

Имя сообщения укажет имя переменной, которая будет доступна в шаблоне и из которой можно будет извлечь *содержимое сообщения*. Также можно указать *список* из нескольких сообщений в виде ассоциативного массива, где ключи элементов станут именами сообщений, а значения элементов — их содержимым.

В следующем примере мы перенаправляем посетителя на главную страницу сайта и создаем всплывающее сообщение `alert` с текстом **Вход не выполнен**:

```
return Redirect::home()->with("alert", "Вход не выполнен");
```

Указание посредников в контроллерах

В *главе 29* мы уже кратко познакомились с посредниками — программными модулями, выполняющимися после завершения работы маршрутизатора и перед выполнением контроллера, а также реализующими различные ключевые задачи, наподобие разграничения доступа. В той же главе мы узнали, что можем указать посредник для определенного маршрута.

Помимо этого, мы можем указать посредник на уровне контроллера — в его конструкторе.

> ***Внимание!***
> В контроллере, созданном средствами прототипирования, конструктор изначально отсутствует. Нам придется создать его самим.

Для указания посредника у контроллера применяется метод `middleware`, вызываемый у самого контроллера:

```
<контроллер>->middleware(<имя посредника>|<список имен посредников>)
```

Имя посредника указывается в виде строки, *список имен посредников* — в виде массива, каждый элемент которого также должен быть строкой.

Метод `middleware` возвращает объект класса `ControllerMiddleware`, представляющий указанный в его вызове посредник.

Листинг 30.3 показывает фрагмент кода контроллера, в конструкторе которого указан посредник `auth`, реализующий разграничение доступа.

Листинг 30.3

```
class CategoryController extends Controller {
  public function __construct() {
```

```
    $this->middleware("auth");
  }
  . . .
}
```

Заданный таким образом посредник будет выполняться перед вызовом любого действия контроллера. Однако мы можем ограничить перечень действий, подпадающих под его влияние, вызвав один из двух приведенных далее методов. Эти методы вызываются у объекта класса `ControllerMiddleware`, возвращенного методом `middleware`.

- `only(`*<имя действия>*`|`*<список имен действий>*`)` — указывает посреднику выполняться только для заданных в его вызове действий (для всех остальных действий контроллера этот посредник выполняться не будет). *Имя действия* указывается в виде строки, *список имен действий* — в виде массива, каждый элемент которого также должен быть строкой.
- `except(`*<имя действия>*`|`*<список имен действий>*`)` — указывает посреднику, наоборот, не выполняться для заданных в его вызове действий (для всех остальных действий контроллера этот посредник будет выполняться). Аргументы этого метода указываются в том же формате, как и в случае метода `only`.

Листинг 30.4 показывает фрагмент кода контроллера, указывающего посреднику `auth` выполняться при вызове всех действий, кроме `index` и `show`.

Листинг 30.4

```
class CategoryController extends Controller {
  public function __construct() {
    $this->middleware("auth")->except(["index", "show"]);
  }
  . . .
}
```

Особые разновидности контроллеров

Для специфических случаев Laravel поддерживает две особые разновидности контроллеров. Рассмотрим их.

Контроллеры-функции

Контроллер-функция может пригодиться, если код, выполняющий обработку данных перед их выводом, настолько мал, что нет смысла создавать из-за него отдельный класс контроллера. Такой контроллер представляет собой обычную функцию, которая указывается прямо в коде маршрутизатора, в выражении задания маршрута во втором аргументе используемого для этого метода — вместо имени действия (о задании маршрутов говорилось в *главе 29*).

Пример контроллера-функции, записанного в выражении создания маршрута в файле web.php:

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

Этот контроллер при обращении к интернет-адресу / выводит на экран страницу, определяемую шаблоном home.blade.php.

Контроллер-функция может принимать в качестве параметров значения, взятые из интернет-адреса, равно как и объект класса `Request`, представляющий запрос. Пример:

```
use Illuminate\Http\Request;
. . .
Route::get("/categories/{id}", function (Request $request, $id) {
  . . .
});
```

Контроллеры-действия

Контроллер-действие — это обычный контроллер, реализованный в виде класса, но содержащий всего одно действие. Это действие реализуется в виде метода с именем `__invoke`.

Листинг 30.5 представляет код контроллера-действия `CategoryList`, выводящего список категорий.

Листинг 30.5

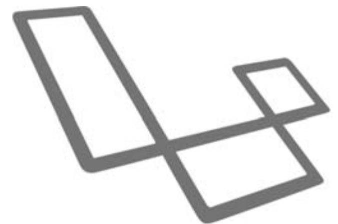
```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;

class CategoryList extends Controller {
  public function __invoke() {
    // Выводим список категорий
  }
}
```

В выражении создания маршрута, в качестве второго аргумента соответствующего метода, указывается только имя такого контроллера:

```
Route::get("/categories", "CategoryList");
```

Метод `__invoke`, реализующий единственное действие, также может принимать в качестве параметров значения, извлеченные из интернет-адреса, и объект класса `Request`, представляющий запрос.



ГЛАВА 31

Шаблоны

Уф! Наконец-то данные извлечены из базы, обработаны, и настала пора вывести их на экран в виде динамической Web-страницы. Как ее сгенерировать? Разумеется, применив шаблон.

Шаблоны Laravel: требования и соглашения

Laravel требует от шаблонов совсем немногого:

- файлы с кодом шаблонов должны иметь расширение blade.php;

  Laravel использует для вывода шаблонов входящий в его состав программный модуль Blade, и это расширение указывает, что такие файлы должны быть обработаны этим модулем;
- файлы с шаблонами должны находиться в папке resources\views папки проекта или вложенной в нее папке.

Принятые соглашения предписывают:

- помещать файлы шаблонов в папки, вложенные в папку resources\views, имена которых совпадают с именами контроллеров, использующих их для вывода данных;
- давать каждому файлу шаблона имя, совпадающее с именем действия, которое использует этот шаблон.

Конечно, следовать этим соглашениям необязательно. Более того, в случае родительских и вложенных шаблонов (о них мы поговорим позже) эти соглашения в любом случае будут нарушены.

Создание шаблонов

Laravel, в отличие от некоторых других фреймворков, не предусматривает никаких средств для прототипирования шаблонов. Поэтому создавать их нам придется полностью вручную.

Шаблон — это, по большему счету, обычная Web-страница, в код которой вставлены инструкции, выполняющие вывод данных. Эти данные хранятся в переменных, чьи имена были указаны в вызовах функции `view` и метода `with`, описанных в *главе 30*, — это обычные переменные, к которым можно обратиться, чтобы вывести хранящиеся в них значения.

Рассмотрим такой код:

```
public function index() {
  $cats = Categories::all();
  return view("categories.index", ["cats" => $cats]);
}
```

Это действие контроллера выполнит вывод данных с применением шаблона index.blade.php, хранящегося в папке categories папки resources\views, и создаст в шаблоне переменную `cats`, в которой будет храниться список категорий.

В коде шаблона для вывода данных можно использовать выражения, написанные на языке PHP. Листинг 31.1 показывает полный код шаблона index.blade.php, выводящего список категорий и использующего для этого обычные языковые конструкции PHP (фрагменты PHP-кода выделены полужирным шрифтом).

Листинг 31.1

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Категории</title>
  </head>
  <body>
    <table>
      <tr>
        <th>ID</th>
        <th>Название</th>
        <th>Описание</th>
        <th>Порядок</th>
      </tr>
      <?php foreach ($cats as $cat) { ?>
        <tr>
          <td><?php echo $cat->id; ?></td>
          <td><?php echo $cat->name; ?></td>
          <td><?php echo $cat->description; ?></td>
          <td><?php echo $cat->order; ?></td>
        </tr>
      <?php } ?>
    </table>
  </body>
</html>
```

Однако гораздо удобнее использовать команды встроенного языка, поддерживаемого программным модулем Blade. Они позволяют несколько сократить объем необходимого PHP-кода и, вообще, сделать код шаблонов более простым для восприятия и сопровождения.

Давайте посмотрим на листинг 31.2. Он показывает полный код того же шаблона, но написанный с применением команд встроенного в Blade языка (эти команды выделены полужирным шрифтом), — как видим, такой код лучше читается.

Листинг 31.2

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Категории</title>
  </head>
  <body>
    <table>
      <tr>
        <th>ID</th>
        <th>Название</th>
        <th>Описание</th>
        <th>Порядок</th>
      </tr>
      @foreach ($cats as $cat)
        <tr>
          <td>{{ $cat->id }}</td>
          <td>{{ $cat->name }}</td>
          <td>{{ $cat->description }}</td>
          <td>{{ $cat->order }}</td>
        </tr>
      @endforeach
    </table>
  </body>
</html>
```

При первом использовании шаблона Laravel выполняет преобразование всех команд языка Blade в обычные выражения PHP — этот процесс называется *компиляцией шаблона*. Файлы с преобразованными шаблонами сохраняются в папке storage\framework\views и используются в дальнейшем при выводе данных для повышения производительности. После исправления файла шаблона он будет перекомпилирован автоматически.

Команды языка Blade

Раз уж зашла речь о языке Blade, давайте познакомимся с поддерживаемыми им командами. Их не очень много.

Команды вывода данных

Для вывода данных применяются следующие команды:

- `{{ <значение> }}` — выводит на экран *значение*;
- `{{ <значение 1> or <значение 2> }}` — выводит на экран *значение 1*. Если оно равно `null` или если представляет собой обращение к несуществующей переменной, будет выведено *значение 2*.

  Перед выводом Laravel выполняет обработку значения, заменяя все встречающиеся в нем HTML-теги соответствующими литералами. Это делается для предотвращения внедрения на страницу вредоносного кода. Так что, если значение содержит HTML-код, он будет выведен на страницу как есть;
- `{!! <значение> !!}` — выводит на экран *значение* без всякой обработки.

Ветвления

Доступны две разновидности ветвления:

- ```
  @if (<условие 1>)
      <блок if 1>
    [@elseif (<условие 2>)
      <блок if 2>
    . . .
    @elseif (<условие n>)
      <блок if n>]
    . . .
    [@else
      <блок else>]
    @endif
  ```

  Это аналог ветвления JavaScript (см. *главу 16*);
- ```
  @unless (<условие>)
      <блок unless>
    @endunless
  ```

  Если *условие* не выполняется, будет выполнен *блок unless*. Если *условие* выполняется, ничего не произойдет.

Пример:

```
@if (count($cats) == 1)
  <p>Одна категория</p>
```

```
@elseif (count($cats) > 1)
  <p>Много категорий</p>
@else
  <p>Список категорий пуст</p>
@endif
```

Циклы

Мы можем использовать следующие разновидности циклов:

- ```
  @for (<выражение инициализации>; <условие>; <приращение>)
      <тело цикла>
    @endfor
  ```

  Это аналог цикла со счетчиком JavaScript;
- ```
  @while (<условие>)
      <тело цикла>
    @endwhile
  ```

  Это аналог цикла с предусловием JavaScript;
- ```
  @foreach (<массив> as <переменная>)
      <тело цикла>
    @endforeach
  ```

  Это аналог цикла по массиву PHP (см. *главу 24*);
- ```
  @forelse (<массив> as <переменная>)
      <тело цикла>
    @empty
      <блок empty>
    @endforelse
  ```

  То же самое, что цикл по массиву, за тем исключением, что, если *массив* пуст (не содержит элементов), будет выполнен *блок empty*.

В следующем примере мы выводим список категорий или, если таковой пуст, соответствующее сообщение:

```
@forelse ($cats as $cat)
  <p>{{ $cat->name }}</p>
@empty
  <p>Список категорий пуст</p>
@endforelse
```

Прерывание и перезапуск цикла

Если Blade поддерживает циклы, то должен иметь поддержку и команд для их прерывания и перезапуска.

- `@break` — оператор прерывания цикла;
- `@break(<условие>)` — оператор, прерывающий цикл при выполнении *условия*;

- `@continue` — оператор перезапуска цикла;
- `@continue(<условие>)` — оператор, перезапускающий цикл при выполнении *условия*.

Листинг 31.3 показывает код, выводящий три категории из списка, начиная с пятой.

Листинг 31.3

```
@for ($i = 0; i < count($cats); i++)
  @continue(i < 5)
  <p>{{ $cats[i]->name }}</p>
  @if (i > 7)
    @break
  @endif
@endfor
```

Служебная переменная *loop*

В теле любого цикла Blade доступна служебная переменная `loop`. Она хранит объект особого класса, представляющий различные сведения о текущем цикле. Свойства этого класса приведены в табл. 31.1.

Таблица 31.1. *Свойства класса, представляющего сведения о цикле Blade*

Свойство	Описание
`index`	Возвращает номер текущего прохода цикла (нумерация начинается с нуля)
`iteration`	Возвращает номер текущего прохода цикла (нумерация начинается с единицы)
`remaining`	Возвращает количество оставшихся проходов цикла (если его можно вычислить)
`count`	Возвращает общее количество проходов цикла (если его можно вычислить)
`first`	Возвращает `true`, если это первый проход цикла, и `false` в противном случае
`last`	Возвращает `true`, если это последний проход цикла, и `false` в противном случае
`depth`	Возвращает уровень вложенности текущего цикла: 1 — для самого «внешнего» цикла, 2 — для вложенного в него цикла, 3 — для цикла, вложенного во вложенный, и т. д.
`parent`	Возвращает аналогичный объект, но хранящий сведения о цикле, в который вложен текущий (если, конечно, текущий цикл вложен в другой)

В листинге 31.4 представлен код, выводящий список категорий с их порядковыми номерами, причем перед списком выводится заголовок, а после него — горизонтальная линия.

Листинг 31.4

```
@foreach ($cats as $cat)
  @if ($loop->first)
    <h1>Категории</h1>
  @endif
  <p>{{ $loop->iteration }} {{ $cat->name }}</p>
  @if ($loop->last)
    <hr>
  @endif
@endforeach
```

Комментарии Blade

Blade позволяет вставить в код шаблона комментарии, которые не будут выведены на экран. Комментарии указываются в следующем формате:

```
{{-- <текст комментария> --}}
```

Пример:

```
{{-- Сейчас будет выведен список статей --}}
```

Вставка PHP-кода

Необходимость во вставке в шаблон PHP-кода возникает нечасто, но все же в некоторых случаях без этого не обойтись. Такие вставки записываются в следующем формате:

```
@php
  <PHP-код>
@endphp
```

Впрочем, автор этой книги предпочитает вставлять в шаблоны фрагменты PHP-кода с помощью привычного тега `<?php . . . ?>`.

Особые случаи вывода данных

Как правило, при разработке шаблонов достаточно знания HTML и встроенного языка Blade. Однако есть случаи, когда понадобятся дополнительные инструменты, и о них стоит сейчас поговорить.

Генерирование интернет-адресов

Для генерирования интернет-адресов предназначаются три функции, которые мы сейчас рассмотрим.

Функция `route` возвращает интернет-адрес, сгенерированный на основе именованного маршрута:

```
route(<имя маршрута>[, <параметры>])
```

Имя маршрута должно совпадать с именем, указанным в вызове метода `name` в выражении, которое создает маршрут (см. *главу 29*). *Параметры* будут подставлены в сформированный на основе маршрута с указанным *именем* интернет-адрес. Если же в маршруте отсутствует параметр с заданным именем, он будет добавлен к интернет-адресу в качестве GET-параметра.

Параметры указываются в виде комбинированного массива. Ключ элемента станет именем параметра, а значение элемента — значением этого параметра. Если в маршруте задействовано внедрение модели (за подробностями — к *главе 29*), соответствующий объект модели можно указать в качестве обычного элемента массива, без ключа.

В следующем примере мы создаем гиперссылку, указывающую на список подкатегорий (маршрут с именем `subcategories.index`), что относятся к текущей категории:

```
<a href="{{ route("subcategories.index",
["category" => $category->id]) }}">Подкатегории</a>
```

Функция `action` возвращает интернет-адрес, указывающий на заданное действие контроллера:

```
action(<действие>[, <параметры>])
```

Действие указывается в формате *<имя контроллера>*`@`*<имя действия>*. *Параметры* задаются в том же формате, что и в случае функции `route`.

Пример:

```
<a href="{{ action("SubcategoryController@index",
["category" => $category->id]) }}">Подкатегории</a>
```

Функция `url`, будучи вызванной без аргументов, возвращает объект класса `UrlGenerator`, с помощью которого можно генерировать различные интернет-адреса, вызывая следующие не принимающие аргументов методы:

- `current` — возвращает относительный интернет-адрес текущей страницы;
- `previous` — возвращает относительный интернет-адрес предыдущей страницы;
- `full` — возвращает полный интернет-адрес текущей страницы.

В следующем примере мы создаем гиперссылку, указывающую на предыдущую страницу:

```
<a href="{{ url()->previous() }}">Назад</a>
```

Если же передать функции `url` в качестве аргумента относительный интернет-адрес (путь к серверной программе):

```
url(<путь>)
```

она вернет созданный на его основе полный интернет-адрес. Пример:

```
<a href="{{ url("/subcategories") }}">Подкатегории</a>
```

Создание Web-форм и элементов управления

Создавать Web-формы и элементы управления, на взгляд автора, удобнее, просто вводя необходимый HTML-код и помещая в нужные места фрагменты, написанные на встроенном языке Blade. Это не так уж и сложно.

Однако здесь есть несколько моментов, о которых обязательно следует знать. Они касаются указания метода отправки данных, защиты от сетевых атак, вывода данных, введенных в форму ранее, и сообщений об ошибках.

Указание метода отправки данных

Web-формы HTML поддерживают только два метода отправки данных: GET и POST — об этом говорилось еще в *главе 6*. Но средства маршрутизации Laravel (см. *главу 29*) также имеют поддержку методов PUT, PATCH и DELETE. Как указать их в форме?

Уж точно не в атрибуте `method` тега `<form>`. Это следует делать в специально созданном в форме скрытом поле, имеющем имя `_method`.

Для создания этого скрытого поля Laravel предоставляет функцию `method_field`:

`method_field(<`*`наименование метода отправки данных`*`>)`

Возвращенное этой функцией значение — HTML-код, создающий это скрытое поле, — следует вставить в код страницы.

Пример:

```
<form action="categories/create" method="post">
  {{ method_field("put") }}
  . . .
</form>
```

Защита от сетевых атак

При выводе каждой страницы Laravel генерирует особый идентификатор, который сохраняет на стороне сервера. Получив отправленные методом POST, PUT, PATCH или DELETE данные, он ищет в них параметр с именем `_token` и сравнивает его значение с сохраненным ранее идентификатором. Обработка данных будет выполнена только в том случае, если оба этих идентификатора совпадают.

Для хранения такого идентификатора в форме создается скрытое поле с именем `_token`. Создать его мы можем, вызвав не принимающую аргументов функцию `csrf_field` и поместив возвращенный ей результат — создающий это скрытое поле HTML-код — в код страницы.

Пример:

```
<form action="categories/create" method="post">
  {{ csrf_field() }}
  . . .
</form>
```

Внимание!

Если не создать в форме скрытое поле с идентификатором, отправленные формой данные не будут обработаны. Более того, мы получим сообщение об ошибке.

Вывод введенных ранее данных

В *главе 32*, посвященной вводу данных, мы будем рассматривать механизм *валидации* — проверки введенных посетителем данных на корректность на стороне сервера. Если данные не пройдут проверку, как правило, выполняется перенаправление на ту же страницу с формой, в которую эти данные были занесены ранее, с подстановкой этих данных в элементы управления формы.

Введенные ранее данные сохраняются на стороне сервера. Они доступны только в момент генерирования страницы с формой, после чего удаляются.

Для вывода таких данных предназначена функция с красноречивым именем `old`:

```
old(<имя элемента управления>[, <значение по умолчанию>])
```

Под *именем элемента управления* подразумевается имя, указанное в атрибуте тега `name`. (Об именах элементов управления говорилось в *главе 6*.) Если значение с таким именем отсутствует, будет выведено *значение по умолчанию* или `null`, если таковое не указано.

В следующем примере мы выводим поле ввода `name`, предназначенное для занесения названия категории. Изначально в него будет подставлено введенное ранее название или, если страница с этой формой открывается впервые (и, следовательно, введенное ранее значение с именем `name` отсутствует), название, взятое из поля `name` модели:

```
<p>Название:<br>
<input name="name" value="{{ old("name", $cat->name) }}"></p>
```

Вывод сообщений об ошибках

Опять же, если посетитель допустил ошибку при вводе данных, разработчик может выполнить перенаправление на страницу с формой, в которую заносились эти данные, с передачей ей сообщений о допущенных ошибках. Так делается практически всегда.

В шаблоне список сообщений об ошибках доступен в виде объекта класса `MessageBag`, хранящегося в переменной `errors`. Для вывода самих сообщений применяются следующие методы этого класса:

- `all` — возвращает массив со всеми сообщениями об ошибках, допущенных при занесении данных во все элементы управления. Не принимает аргументов;
- `get(<имя элемента управления>)` — возвращает массив со всеми сообщениями об ошибках, допущенных при занесении данных только в элемент управления с указанным *именем*;
- `first([<имя элемента управления>])` — возвращает первое сообщение об ошибке. Если указано *имя элемента управления*, возвращает первое сообщение

об ошибке, относящееся к элементу с этим именем. Если ошибок нет, возвращается пустая строка;

- `has(<имя элемента управления>)` — возвращает `true`, если есть хотя бы одно сообщение об ошибке, относящееся к элементу управления с указанным *именем*, и `false` в противном случае;
- `count` — возвращает общее количество сообщений об ошибках в списке. Не принимает аргументов;
- `any` — возвращает `true`, если есть хотя бы одно сообщение об ошибке, и `false` в противном случае. Не принимает аргументов;
- `isEmpty` — то же самое, что и `any`.

Пример кода, выводящего сообщения об ошибках, которые относятся к полю для ввода названия категории (`name`), приведен в листинге 31.5.

Листинг 31.5

```
<p>Название:<br>
<input name="name" value="{{ old("name", $cat->name) }}"></p>
@if ($errors->has("name"))
  <ul>
    @foreach ($errors->get("name") as $error)
      <li>{{ $error }}</li>
    @endforeach
  </ul>
@endif
```

Вывод всплывающих сообщений

О всплывающих сообщениях говорилось в *главе 30*. Для их вывода применяется функция `session`:

```
session(<имя сообщения>)
```

В качестве результата она возвращает текст всплывающего сообщения, который можно вывести на экран.

Пример:

```
<div class="flash">{{ session("alert") }}</div>
```

Вывод пагинатора

В *главе 30* мы познакомились с пагинатором, который выполняет разбиение набора записей на отдельные страницы, и научились создавать его в коде контроллера. Осталось выяснить, как вывести гиперссылки, указывающие на отдельные страницы такого набора записей.

Вывести набор этих гиперссылок очень просто — достаточно вызвать метод `links`, поддерживаемый классами `Paginator` и `LengthAwarePaginator`:

```
<пагинатор>->links([<имя шаблона пагинатора>])
```

Если *имя шаблона пагинатора* не указано, вывод будет выполнен с применением шаблона, встроенного в Laravel.

В следующем примере мы выводим набор гиперссылок, указывающих на отдельные страницы списка статей, с применением встроенного шаблона:

```
<div class="pagination">{{ $articles->links() }}</div>
```

Писать новый шаблон пагинатора довольно трудоемко. Проще создать копию встроенного шаблона и внести в нее соответствующие изменения. Для создания копии встроенного шаблона следует вызвать консоль OpenServer, перейти в папку проекта и набрать команду:

```
php artisan vendor:publish --tag=laravel-pagination
```

После чего будет создана папка resources\views\vendor\pagination, а в нее — четыре файла с шаблонами. Нас более всего интересуют следующие два:

- simple-default.blade.php — шаблон упрощенного пагинатора (класса `Paginator`), выводящий только гиперссылки, которые ведут на следующую и предыдущую страницы;
- default.blade.php — шаблон полнофункционального пагинатора (класса `LengthAwarePaginator`), выводящий полный набор гиперссылок.

Если нужно всего лишь немного изменить внешний вид гиперссылок пагинатора, достаточно исправить код этих шаблонов и вызвать метод `links` без указания аргументов. В таком случае Laravel использует шаблон, хранящийся в папке resources\views\vendor\pagination.

Также можно создать на основе любого из этих шаблонов полностью другой шаблон и использовать для вывода пагинатора именно его. В этом случае следует указать имя этого шаблона в вызове метода `links`.

В следующем примере мы используем для вывода пагинатора шаблон cool-paginator.blade.php, хранящийся в папке resources\views\common:

```
<div class="pagination">{{ $articles->links("common.cool-paginator") }}
</div>
```

В шаблон пагинатора в качестве контекста данных передаются два значения:

- `paginator` — сам пагинатор;
- `elements` — массив, хранящий элементы пагинатора.

Первое значение — сам пагинатор — мы можем использовать, чтобы получить различные сведения о пагинаторе. Так, мы можем вызвать метод `hasPages` (см. *главу 30*), чтобы выяснить, содержит ли пагинатор больше одной страницы, перед тем как вывести гиперссылки пагинатора на экран.

Массив с элементами пагинатора включает элементы двух видов:

- строковые (как правило, символ многоточия) — обозначает часть страниц, которые будут скрыты при выводе. Пагинатор Laravel не выводит на экран часть страниц, если таковых получается слишком много, ради компактности.

  Строковый элемент выводится на экран как есть;
- ассоциативные массивы — содержат интернет-адреса отдельных страниц. Ключами элементов таких массивов являются порядковые номера страниц, а значениями элементов — сами интернет-адреса.

  На основе элементов таких массивов формируются наборы гиперссылок, указывающих на отдельные страницы пагинатора.

Впоследствии, при разработке сайта, мы создадим шаблон пагинатора. Его код можно увидеть в листинге 41.6 — он хорошо иллюстрирует принципы, согласно которым выполняется вывод пагинатора на экран.

Вложенные шаблоны

Часто в различных шаблонах приходится записывать одинаковые фрагменты кода. Это могут быть списки сообщений об ошибках, элементы для вывода всплывающих сообщений и проч.

Чтобы не копировать один и тот же фрагмент кода из одного шаблона в другой, мы можем вынести его в отдельный шаблон. А в код каждого шаблона, в котором должен быть выведен этот фрагмент, мы поставим своего рода ссылку на шаблон с фрагментом.

Шаблоны подобного рода, предназначенные для вывода в составе других шаблонов, в Laravel носят название *вложенных*.

По существующим соглашениям все вложенные шаблоны должны помещаться в папку common папки resources\views. (Отметим, что изначально папка common там отсутствует, и нам придется создать ее самим.) Хотя, разумеется, мы можем поместить их в папку с другим именем или вообще сохранить непосредственно в папке resources\views. В остальном шаблоны такого рода должны удовлетворять требованиям, предъявляемым к обычным шаблонам.

Вложенные шаблоны по умолчанию получают все данные, доступные в шаблонах, в составе которых они выводятся.

Листинг 31.6 показывает код шаблона errors.blade.php, выводящего список всех ошибок, что были допущены при занесении данных в форму.

Листинг 31.6

```
@if ($errors->any())
  <ul>
    @foreach ($errors->all() as $error)
      <li>{{ $error }}</li>
```

```
    @endforeach
  </ul>
@endif
```

Для вывода вложенного шаблона используется команда `@include`:

```
@include(<имя вложенного шаблона>
[, <значения, добавляемые к контексту данных вложенного шаблона>])
```

Имя вложенного шаблона задается в виде строки. Оно должно указываться в том же формате, что применяется для указания шаблонов в вызове функции `view` в действиях контроллеров (см. *главу 30*). Если шаблон с таким именем не существует, команда выдаст ошибку.

Значения, добавляемые к контексту данных вложенного шаблона, оформляются в виде ассоциативного массива. Здесь применяется тот же формат, что и в вызове функции `view`.

В следующем примере мы выводим на экран вложенный шаблон errors.blade.php из папки common:

```
@include("common.errors")
```

Команда `@includeIf` имеет такой же формат вызова и выполняет ту же задачу, но в случае отсутствия шаблона с указанным именем ничего не делает.

Команда `@each` выполняет вывод вложенного шаблона для каждого из элементов указанного массива:

```
@each(<имя вложенного шаблона>, <массив>,
<имя переменной с элементом массива, передаваемой вложенному массиву>
[, <имя вложенного шаблона, выводящегося, если массив пуст>])
```

Имя переменной с элементом массива, передаваемой вложенному шаблону, указывается в виде строки. Очередной элемент *массива* будет присвоен этой переменной, затем она будет передана вложенному шаблону с указанным *именем* в качестве контекста данных. Именно из этой переменной вложенный шаблон будет брать выводимое значение.

Если *имя вложенного шаблона, выводящегося, если массив пуст*, не указано, то в случае отсутствия у *массива* элементов команда `@each` ничего не делает.

Пример:

```
{{-- Код вложенного шаблона cat.blade.php --}}
<p>{{ @cat }}</p>

{{-- Вывод вложенного шаблона для списка категорий $cats --}}
@each("common.cat", $cats, "cat")
```

Наследование шаблонов

Когда мы начнем создавать шаблоны для сайта, то вскоре выясним одну неприятную вещь. Все эти шаблоны будут содержать довольно большой объем повторяющегося кода: теги секции заголовка, теги, формирующие разметку страниц, код «шапки», панели навигации, «поддона» и проч.

Возникает вполне очевидное решение этой проблемы:

- ❒ создать шаблон, содержащий весь повторяющийся код;
- ❒ в коде этого шаблона пометить места, куда будет вставляться содержание, уникальное для каждой страницы;
- ❒ во всех прочих шаблонах обозначить фрагменты кода, которые должны быть вставлены в помеченные места созданного ранее шаблона.

Все это с легкостью реализуется в Laravel благодаря поддерживаемому этим фреймворком механизму, называемому *наследованием шаблонов*.

Создание шаблонов-родителей

Шаблон, от которого наследуются все остальные шаблоны, по аналогии с классами PHP называется *родителем*. Он создается с применением тех же приемов, что и обычный шаблон.

Как уже говорилось, в коде шаблона-родителя следует пометить места, куда будет вставлено уникальное для каждой страницы содержание. Такое место в терминологии Laravel носит название *секции шаблона*.

Секция шаблона создается командой встроенного языка Blade под названием `@yield`:

```
@yield(<имя секции>)
```

Имя секции используется для того, чтобы впоследствии, при указании уникального содержания, сослаться на эту секцию. Оно задается в виде строки.

Пример кода шаблона-родителя main.blade.php с двумя секциями представлен в листинге 31.7. Секция `title` служит для добавления текста в название страницы, а секция `main` — для вставки на страницу основного содержания. (Код, создающий эти секции, выделен полужирным шрифтом.)

Листинг 31.7

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>@yield("title") - Сайт публикаций</title>
  </head>
```

```
  <body>
    @yield("main")
  </body>
</html>
```

Принятые в Laravel соглашения предписывают помещать файлы с шаблонами-родителями в папку layouts папки resources\views (хотя это и необязательно). В остальном шаблоны такого рода должны удовлетворять требованиям, предъявляемым к обычным шаблонам.

Создание шаблонов-потомков

Опять же, по аналогии с классами PHP, шаблоны, задающие уникальное для страницы содержание и основанные на каком-либо шаблоне-родителе, носят название *потомков*.

Первое, что следует указать в коде шаблона-потомка, — имя шаблона-родителя, на основе которого создан шаблон. Для этого служит команда `@extends`:

```
@extends(<имя шаблона-родителя>)
```

Имя шаблона-родителя задается в виде строки. Оно должно быть записано в том же формате, что применяется для указания шаблонов в вызове функции `view` в действиях контроллеров (см. *главу 30*).

Еще раз отметим, что команда `@extends` должна находиться в самом начале кода шаблона-потомка.

Для обозначения содержания, которое должно быть помещено в одну из секций шаблона-родителя, применяется команда `@section`. Она поддерживает два формата вызова:

- `@section(<имя секции>, <выводимое значение>)` — выводит в секцию с указанным *именем* заданное *значение*. Применяется, если в секцию следует вставить единичное значение (строку, число и т. п.);
- ```
  @section(<имя секции>)
    <вставляемый в эту секцию код>
  @endsection
  ```

  Выводит в секцию с указанным *именем* целый фрагмент *кода*.

Листинг 31.8 показывает код шаблона-потомка categories\index.blade.php, основанного на ранее созданном шаблоне (предполагается, что он был сохранен в файле main.blade.php и помещен в папку layouts).

Листинг 31.8

```
@extends("layouts.main")
@section("title", "Категории")
@section("main")
```
```

```
  <table>
    <tr>
      <th>ID</th>
      <th>Название</th>
      <th>Описание</th>
      <th>Порядок</th>
    </tr>
    @foreach ($cats as $cat)
      <tr>
        <td>{{ $cat->id }}</td>
        <td>{{ $cat->name }}</td>
        <td>{{ $cat->description }}</td>
        <td>{{ $cat->order }}</td>
      </tr>
    @endforeach
  </table>
@endsection
```

При выводе данных в коде действия контроллера, в вызове функции `view` указывается имя шаблона-потомка:

```
return view("categories.index", ["cats" => $cats]);
```

Стеки

Стек чем-то напоминает секцию шаблона-родителя. Отличается он двумя особенностями:

- содержание, указанное для стека в каком-либо шаблоне, будет добавлено к уже имеющемуся в стеке (а не заменит помещенное в него ранее, как происходит в таком случае у секции);
- мы можем добавлять содержание в стек неоднократно и в разных шаблонах (родителе, потомке и даже вложенном).

Стек создается с помощью команды `@stack`:

```
@stack(<имя стека>)
```

Имя стека указывается в виде строки.

Для добавления содержания в стек служит команда следующего вида:

```
@push(<имя стека>)
  <добавляемый в этот стек код>
@endpush
```

Пример использования стека можно увидеть в листинге 31.9.

Листинг 31.9

```
{{-- Создаем в секции заголовка шаблона-родителя стек head --}}
<head>
  @stack("head")
</head>

{{-- В шаблоне-потомке добавляем к этому стеку теги названия
и Web-сценария --}}
@push("head")
  <title>Статьи</title>
  <script src="/main.js"><script>
@endpush

{{-- Во вложенном шаблоне добавляем к этому стеку тег таблицы стилей --}}
@push("head")
  <link rel="stylesheet" href="/main.css" type="text/css">
@endpush
```

В результате сгенерированная страница будет иметь следующую секцию заголовка:

```
<head>
  <title>Статьи</title>
  <script src="/main.js"><script>
  <link rel="stylesheet" href="/main.css" type="text/css">
</head>
```

Разделяемые данные и составители

Часто случается, что несколько шаблонов должны выводить один и тот же набор данных. Это может быть, например, колонка новостей, присутствующая на нескольких разных страницах.

А еще чаще панель навигации, помещенная в шаблон-родитель, формируется на основе данных, которые берутся из информационной базы. Так, на сайте электронных публикаций в составе этой панели навигации придется выводить список категорий.

Если пользоваться для добавления таких значений в контекст данных шаблона средствами, описанными в *главе 30*, нам придется вставлять один и тот же код в каждое действие каждого контроллера. Но есть способ лучше. Даже два.

Разделяемые данные

Если какое-то значение должно выводиться сразу во всех шаблонах, наилучшее решение — поместить это значение в состав *разделяемых данных* Laravel. Такие данные будут доступны во всех без исключения шаблонах, что имеются в составе проекта.

Поместить значение в состав разделяемых данных можно вызовом метода `share` фасада `View`:

```
View::share(<имя значения>, <значение> | <список значений>)
```

Можно указать либо два аргумента: *имя значения* и само *значение*, либо один аргумент — *список значений*, представленный в виде ассоциативного массива, который формируется по тем же правилам, что массив, передаваемый функции `view`.

Вызов метода `share` помещается в метод `boot` класса провайдера `AppServiceProvider` (о нем рассказывалось в *главе 28*).

Листинг 31.10 показывает пример установки в составе разделяемых данных названия сайта, списков категорий и пользователей.

Листинг 31.10

```
use Illuminate\Support\Facades\View;
use App\Category;
use App\User;
class AppServiceProvider extends ServiceProvider {
  public function boot() {
    View::share("siteHeader", "Сайт публикаций");
    View::share(["cats" => Category::all(), "users" => User::all()]);
  }
  . . .
}
```

Значение, помещенное в состав разделяемых данных, будет доступно в шаблоне через переменную с именем, совпадающим с именем, что было указано в вызове метода `share`, — как и в случае со значениями, помещенными в контекст данных. Пример:

```
<title>{{ $siteHeader }}</title>
```

Составители

Составители будут наиболее полезны в случае, если необходимо сделать данные доступными только в некоторых шаблонах.

Составитель представляет собой отдельный класс и, как следствие, располагается в отдельном файле. Обычно составители помещаются в пространство имен `App\Http\ViewComposers`, а хранящие их файлы — в папку app\Http\ViewComposers. Отметим, что папка ViewComposers в папке Http изначально не существует, и нам самим придется создать ее.

В классе составителя объявляется публичный метод `compose`, в качестве единственного параметра принимающий объект класса `View`, расположенного в пространстве имен `Illuminate\View`, который представляет шаблон. В теле этого метода выпол-

няется добавление значения к контексту данных этого шаблона путем вызова знакомого нам по *главе 30* метода `with`.

Листинг 31.11 показывает код класса-составителя `MainComposer`, который добавляет в контекст данных шаблона список категорий.

Листинг 31.11

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

use Illuminate\View\View;
use App\Category;

class MainComposer {
  public function compose(View $view) {
    $view->with("cats", Category::all());
  }
}
```

Вновь созданный составитель должен быть зарегистрирован во фреймворке. Для этого применяются два метода фасада `View`:

- `composer(<имя шаблона>|<список имен шаблонов>, <полное имя класса составителя>)` — регистрирует составитель с указанным в виде строки *полным именем* либо для шаблона с указанным *именем*, либо для всех шаблонов, перечисленных в *списке* (который должен представлять собой массив).

  Если нужно зарегистрировать какой-либо составитель сразу для всех шаблонов, в качестве первого аргумента методу `composer` нужно передать строку `"*"`;

- `composers(<список составителей и шаблонов>)` — регистрирует несколько составителей для различных шаблонов. Передаваемый в качестве аргумента *список* должен представлять собой ассоциативный массив, ключи элементов которого являются полными именами составителей, а значения элементов — либо именами шаблонов, либо массивами с именами шаблонов.

  Вместо имени шаблона можно указать строку `"*"` — тогда этот составитель будет зарегистрирован сразу для всех шаблонов.

Любопытно, что зарегистрировать составитель мы можем не только для обычного шаблона, но и для родительского, задающего разметку страницы, и даже для вложенного. Эта особенность Laravel пригодится нам при разработке сайта.

Вызовы обоих этих методов указываются в теле метода `boot` провайдера `AppServiceProvider`.

Листинг 31.12 показывает пример регистрации составителя `MainComposer` для всех шаблонов проекта.

Листинг 31.12

```
. . .
use Illuminate\Support\Facades\View;
class AppServiceProvider extends ServiceProvider {
  public function boot() {
    View::composer("*", "App\Http\ViewComposers\MainComposer");
  }
  . . .
}
```

Получение доступа к контроллеру

Иногда бывает необходимо вывести на странице значение, сохраненное в свойстве контроллера. Но как в этом случае в шаблоне получить объект, представляющий текущий контроллер? Очень просто:

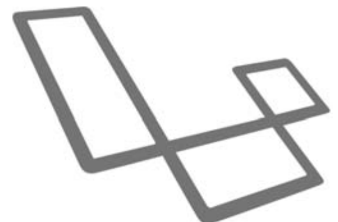
- сначала нужно получить объект класса `Route`, представляющий текущий маршрут. Для этого служит метод `route` класса `Request`, в таком случае вызываемый без аргументов и возвращающий объект маршрута.

  Получить объект класса `Request`, представляющий запрос, в коде шаблона удобнее всего вызовом функции `request`, которая была описана в *главе 30*;
- у полученного ранее объекта класса `Route` следует вызвать не принимающий аргументов метод `getController`. Он вернет объект текущего контроллера.

Вот пример вывода на экран значения, сохраненного в свойстве `pageTitle` контроллера:

```
{{ request()->route()->getController()->pageTitle }}
```

Кстати, хранение данных, которые требуется сделать доступными для нескольких или всех шаблонов, в свойствах контроллера — неплохая альтернатива использованию разделяемых данных и составителей, описанных ранее. Только эти свойства следует объявлять в классе `Controller` (о нем говорилось в *главе 30*) — родителе всех классов контроллеров. В результате все контроллеры унаследуют эти свойства у родителя.



ГЛАВА 32

Ввод и правка данных

Помимо вывода данных, в динамических сайтах обязательно следует предусмотреть средства для их ввода и правки. В самом деле, нужно же наполнить сайт данными: категориями, подкатегориями, статьями и зарегистрированными пользователями. Да и дать посетителям возможность комментировать статьи тоже нужно — куда же без этого!

Кроме того, следует предусмотреть возможность выгрузки на сайт файлов. Ведь пользователи нашего сайта: авторы, редакторы и администраторы — будут вставлять в текст своих статей иллюстрации и готовить для них дополнительные материалы для загрузки.

Создание, правка и удаление отдельных записей

Реализовать ввод, правку и удаление записей на основе данных, введенных в Web-форму, проще всего средствами, предоставляемыми соответствующей моделью.

Создание и правка записей

Для создания и правки записей удобнее использовать одну страницу с формой и создать в контроллере два действия, первое из которых будет выводить эту страницу, а второе — выполнять сохранение новой или измененной записи на основе занесенных в форму данных.

Этот подход отличается от предлагаемого Laravel, когда для создания и правки записи применяются разные действия. Он гораздо удобнее, так как нам не придется писать в разных действиях и шаблонах один и тот же код.

Создание Web-формы для ввода и правки записи

Сначала следует подготовить форму, с помощью которой посетитель будет создавать и править записи.

Создавать форму нужно, следуя приведенным далее правилам.

- В теге `<form>` самой формы, в атрибуте `action`, надо указать интернет-адрес, указывающий на действие контроллера, которое будет выполнять создание и изменение записи.
- В форме нужно создать скрытое поле, хранящее идентификатор, который участвует в предотвращении сетевых атак.
- Если в форме будет выполняться правка записи, в нее следует поместить скрытое поле с обозначением метода отправки данных PUT или PATCH.
- Опять же, если в форме будет выполняться правка записи, в нее надо поместить скрытое поле, хранящее номер записи. Это нужно для того, чтобы действие контроллера смогло найти исправляемую запись.

  Здесь можно использовать тот факт, что у создаваемой записи поле номера (`id`) пусто, а у исправляемой — нет (в этом случае оно хранит само значение номера). Следовательно, если поле номера не пусто, то в форму следует поместить скрытые поля с обозначением метода отправки данных и номером записи.
- В элементах управления в качестве значения, выводимого изначально, следует указать старое значение, полученное вызовом функции `old` (за подробностями — к *главе 31*) или, если таковое отсутствует (то есть, если страница с формой выводится впервые), значение из соответствующего поля модели.

  Это пригодится чуть позже, когда мы будем реализовывать проверку занесенных в форму данных на корректность на стороне сервера.
- Что касается имен элементов управления, задаваемых атрибутами `name` соответствующих тегов, то здесь нужно руководствоваться следующими соображениями:
  - если имена элементов управления совпадают с именами соответствующих полей модели, мы сможем выполнить создание или исправление записи самым простым способом (как именно, будет сказано далее);
  - если же имена элементов управления отличаются от имен полей, для создания или исправления записи нам придется написать несколько более сложный код.

  Поэтому лучше делать имена элементов управления такими же, как и имена модели (конечно, если позволяет ситуация).

Листинг 32.1 показывает код формы, предназначенной для создания и правки категорий, отправляющей занесенные данные действию `save` контроллера `CategoryController`, входящей в состав шаблона categories\input.blade.php и написанной с учетом всего сказанного.

Листинг 32.1

```
<form action="{{ action('CategoryController@save') }}"
method="POST">
```

```
  @if ($cat->id)
    {{ method_field('PUT') }}
    <input type="hidden" name="id" value="{{ old("id", $cat->id) }}">
  @endif
  {{ csrf_field() }}
  <p>Название:<br>
  <input name="name" value="{{ old("name", $cat->name) }}"></p>
  <p>Описание:<br>
  <textarea name="description">{{ $cat->description }}</textarea></p>
  <p>Порядок:<br>
  <input name="order" value="{{ $cat->order }}"></p>
  <p><input type="submit" value="Сохранить"></p>
</form>
```

Если в форме для указания значения для поля логического типа предполагается использовать флажок, обязательно следует задать для него в качестве значения 1 (для этого, как мы помним из *главы 6*, используется атрибут `value` тега `<input>`). В противном случае у нас не получится выполнить сохранение записи с применением описанного далее приема. Пример:

```
<input type="checkbox" name="hidden" value="1"
@if (old('hidden', $comment->hidden)) checked @endif>
```

Отметим, что если поле, значение которого будет задаваться с применением этого флажка, хранит `true`, мы подставляем в создающий флажок тег `<input>` атрибут `checked`. Таким образом мы сделаем флажок изначально установленным.

Создание записи

Созданием записи будут «заведовать» два действия. Первое выполнит создание новой «пустой» записи и вывод на экран страницы с формой, второе — получение занесенных в форму данных и, собственно, сохранение записи. Первое действие будет выполняться при отправке данных методом GET, второе — при отправке данных методом POST. Первое действие можно назвать `input`, второе — `save` или `store`.

Сначала следует написать маршруты для обоих этих действий. Они могут выглядеть так:

```
Route::get("/categories/create", "CategoryController@input")
->name("categories.create");
Route::post("/categories", "CategoryController@save");
```

Теперь можно создать сами действия. Первое — `input`:

- проверит, был ли ему передан в качестве параметра номер записи, и выполнит следующие действия только в том случае, если он не был передан;
- создаст «пустой» объект класса модели — «пустую» запись. Это выполняется обычным путем, с помощью оператора `new`, без указания каких-либо аргументов. Пример:

  ```
  $cat = new Category;
  ```

Нужно, однако, отметить, что при создании новой записи можно указать в качестве аргумента ассоциативный массив. Ключи элементов этого массива укажут имена полей, а значения элементов — значения, которые будут подставлены в эти поля сразу при создании записи.

В следующем примере мы создаем новую категорию, в поле `name` которой будет подставлено значение `"Игры"`, а в поле `order` — значение `10`:

```
$cat = new Category(["name" => "Игры", "order" => 10]);
```

- выведет на экран страницу с формой, созданной на основе подготовленного ранее шаблона;
- передаст этому шаблону в составе контекста данных только что созданный объект модели — «пустую» запись.

Пример кода этого действия показан в листинге 32.2. Отметим, что его параметр, принимающий номер записи, сделан необязательным.

Листинг 32.2

```
public function input($id = null) {
  if ($id) {
    // Здесь будет написан код, выводящий форму для правки записи
  } else {
    $cat = new Category;
  }
  return view("categories.input", ["cat" => $cat]);
}
```

Второе действие — `save`:

- проверит, присутствует ли в составе данных, переданных формой, номер записи (поле `id`), и выполнит последующие шаги только в том случае, если он отсутствует;
- выполнит создание новой записи;
- произведет перенаправление на страницу списка записей (как правило, перенаправление производится именно на эту страницу).

Для сохранения новой записи можно выбрать один из двух описанных далее способов.

Способ первый — сохранение записи вызовом статического метода `create` модели:

```
<класс модели>::create([<данные>])
```

Данные указываются в виде ассоциативного массива, в котором ключи элементов выступают в качестве имен полей модели, а значения элементов станут значениями, заносимыми в эти поля. Если при создании формы мы указали имена элементов управления, совпадающие с именами полей (как рекомендовалось ранее), этот массив можно получить вызовом метода `all` класса `Request` (см. *главу 30*).

> **Внимание!**
>
> Метод `create` использует массовое присваивание значений соответствующим им полям модели. Чтобы оно выполнялось, в классе модели нужно указать либо список полей, вовлеченных в массовое присваивание, либо список полей, не вовлеченных в него. Как это сделать, описывалось в *главе 28*.

Метод `create` немедленно создает, сохраняет новую запись и в качестве результата возвращает представляющий ее объект класса модели.

Пример кода действия `save`, в котором используется этот метод, показан в листинге 32.3.

Листинг 32.3

```
public function save(Request $request) {
  if ($request->has("id")) {
    // Здесь будет записан код, выполняющий исправление записи
  } else {
    Category::create($request->all());
  }
  return redirect()->action("CategoryController@index");
}
```

Способ второй — создание «пустого» объекта класса модели, занесение в его поля значений вручную и последующее сохранение.

Создать «пустой» объект класса модели можно все тем же оператором `new`. Занести в модель значения можно, обращаясь к свойствам, которые соответствуют полям модели (об этих полях рассказывалось в *главе 30*).

А для сохранения записи используется не принимающий аргументов метод `save` модели. Этот метод в качестве результата возвращает `true`, если сохранение выполнилось успешно, и `false` в противном случае.

Второй способ применяется, если имена элементов управления в форме не совпадают с именами полей модели или если нужно создать новую запись на основании данных, не занесенных в форму, а сгенерированных программно.

Пример реализации второго подхода показан в листинге 32.4.

Листинг 32.4

```
public function save(Request $request) {
  if ($request->has("id")) {
    // Здесь будет записан код, выполняющий исправление записи
  } else {
    $cat = new Category();
    $cat->name = $request->name;
    $cat->description = $request->description;
```

```
    $cat->order = $request->order;
    $cat->save();
  }
  return redirect()->action("CategoryController@index");
}
```

Нужно иметь в виду, что, если какое-либо значение в форму заносится с применением флажка, оно не будет сохранено в записи в обоих случаях. Нам придется написать дополнительный код, который сохранит это значение явно.

Мы можем использовать тот факт, что в случае установки флажка в составе запроса будет передан GET-параметр с именем, совпадающим с именем флажка, — оно, как мы помним, указывается в атрибуте `name` тега `<input>`, — и значением, которое указывается в атрибуте `value` того же тега и в качестве которого следует указать число 1 (как мы сделали это ранее). Если же флажок сброшен, таковой GET-параметр вообще будет отсутствовать в запросе. Следовательно, нам достаточно проверить, присутствует ли этот GET-параметр в запросе, и, если присутствует, занести в поле модели `true`, в противном случае — `false`.

Пример:

```
$comment = new Comment($request->all());
$comment->hidden = ($request->has("hidden")) ? true : false;
$comment->save();
```

Останется лишь создать гиперссылку, ведущую на страницу с формой создания новой записи. Формирующий ее код может быть таким:

```
<a href="{{ route("categories.create") }}">Добавить категорию</a>
```

Правка записи

Правка записи также выполняется двумя действиями контроллера. Первое действие мы уже почти написали — это действие `input`, в котором нам понадобится добавить совсем немного кода. В нашем случае оно будет искать в таблице запись, выводить на экран страницу с формой, сгенерированную на основе того же шаблона, и передавать найденную запись этому шаблону в составе контекста данных. Второе действие — также написанное нами ранее `save` (или `store`), которое мы немного дополним, — будет вызываться при отправке данных методом PUT (или PATCH) и выполнит изменение записи на основе введенных в форму данных.

Соответствующие этим действиям маршруты будут такими:

```
Route::get("/categories/{id}/edit", "CategoryController@input");
Route::put("/categories", "CategoryController@save");
```

Для поиска записей в первом действии (`input`) можно использовать методы `find` или `findOrFail` модели, описанные в *главе 30*.

Полный код действия `input`, выполняющего вывод формы и для создания новой записи, и для правки уже существующей, показан в листинге 32.5.

Листинг 32.5

```
public function input($id = null) {
  if ($id) {
    $cat = Category::findOrFail($id);
  } else {
    $cat = new Category;
  }
  return view("categories.input", ["cat" => $cat]);
}
```

Во втором действии (`save` или `store`) после выполнения поиска записи (для чего применяются те же методы `find` и `findOrFail` модели) следует заполнить ее новыми данными. Для этого проще всего вызвать метод `fill` модели:

<модель>`->fill(`*<данные>*`)`

Данные здесь указываются в виде ассоциативного массива, аналогичного тому, что передается описанному ранее методу `create`. В качестве результата метод `fill` возвращает тот же объект класса модели, у которого был вызван.

Сохранение обновленной записи производится вызовом все того же метода `save`.

Листинг 32.6 показывает полный код действия `save`, выполняющего и создание, и обновление записи.

Листинг 32.6

```
public function save(Request $request) {
  if ($request->has("id")) {
    $cat = Category::findOrFail($request->id);
    $cat->fill($request->all())->save();
  } else {
    Category::create($request->all());
  }
  return redirect()->action("CategoryController@index");
}
```

И, наконец, можно найти запись, вручную занести в ее поля полученные от формы данные и вызвать метод `save` (листинг 32.7).

Листинг 32.7

```
public function save(Request $request) {
  if ($request->has("id")) {
    $cat = Category::findOrFail($request->id);
  } else {
    $cat = new Category();
  }
```

```
  $cat->name = $request->name;
  $cat->description = $request->description;
  $cat->order = $request->order;
  $cat->save();
  return redirect()->action("CategoryController@index");
}
```

Если в форме применяется флажок, придется написать дополнительный код, который выполнит сохранение занесенного с помощью флажка значения. Этот код аналогичен представленному ранее.

Код, создающий гиперссылку, что ведет на страницу с формой для правки записи, может быть таким:

```
@foreach ($cats as $cat)
  . . .
  <a href="{{ action("CategoryController@input",
  ["id" => $cat->id]) }}">Исправить категорию</a>
@endforeach
```

Удаление записей

Процедура удаления записи, предлагаемая разработчиками Laravel, также не очень удобна, поскольку требует создания отдельной страницы с формой, в которую помещается кнопка удаления. Лучше применить другой подход: создать действие с именем `destroy` или `delete`, которое будет вызываться при отправке данных методом GET на особый интернет-адрес, выполнять все действия по удалению записей, после чего перенаправлять посетителя на страницу списка записей.

Для этого нужно написать вот такой маршрут:

```
Route::get("/categories/{id}/delete", "CategoryController@destroy");
```

Удалить запись проще всего вызовом статического метода `destroy` модели:

```
<класс модели>::destroy(<номер удаляемой записи>|
<список номеров удаляемых записей>)
```

Как видим, можно удалить как одну запись, передав методу ее номер, так и сразу несколько, указав массив с номерами. В качестве результата возвращается количество действительно удаленных записей.

Вот пример кода действия, выполняющего удаление записи:

```
public function destroy($id) {
  Category::destroy($id);
  return redirect()->action("CategoryController@index");
}
```

Еще можно использовать не принимающий аргументов метод `delete` модели. Он удаляет запись, представляемую объектом класса модели, у которого был вызван, и

возвращает `true`, если удаление было выполнено успешно, и `false` в противном случае. Пример:

```
public function destroy($id) {
  $cat = Category::findOrFail($id);
  $cat->delete();
  return redirect()->action("CategoryController@index");
}
```

Код, создающий гиперссылку, ведущую на удаляющее запись действие, может быть таким:

```
@foreach ($cats as $cat)
  . . .
  <a href="{{ action("CategoryController@destroy",
  ["id" => $cat->id]) }}">Удалить категорию</a>
@endforeach
```

Обработка связей между таблицами

Мы только что рассмотрели простейшие случаи вывода форм для создания и правки записей. Однако часто запись, выводящаяся в форме, связана с другими записями с применением связи «один-ко-многим» или даже «многие-ко-многим». Как быть в таком случае?

Связь «один-ко-многим»

Рассмотрим сначала случай, когда в форме следует вывести запись вторичной таблицы, связанную с записью первичной таблицы связью «один-ко-многим». В этом случае нужно поместить в форму список, обычный или раскрывающийся, перечисляющий все записи первичной таблицы, — в этом списке посетитель сможет выбрать запись, с которой будет связана создаваемая или исправляемая им. Типичный пример — подкатегория, в ее случае нужно вывести список категорий, в котором посетитель выберет ту, к которой относится создаваемая или исправляемая подкатегория.

Сделать это несложно. Далее приведена последовательность действий, которые следует выполнить.

- В действии контроллера, которое будет выводить страницу с формой, следует подготовить перечень записей первичной таблицы, которые будут выводиться в списке этой формы. Сделать это можно, применяя способы, которые уже знакомы нам по *главе 30*. Такой перечень должен быть передан шаблону с формой в составе контекста данных.

  В следующем примере в коде действия, выводящего страницу с формой, мы готовим перечень категорий и передаем его шаблону с формой ввода подкатегории вместе с самой подкатегорией (создаваемой или исправляемой):

  ```
  $cats = Category::orderBy("order", "desc")->orderBy("name")->get();
  return view("subcategories.input", ["subcat" => $subcat,
  "cats" => $cats]);
  ```

- В шаблоне записывается код, который создает список, показывающий записи первичной таблицы. В цикле из полученного перечня записей первичной таблицы извлекаются отдельные записи, и на основе каждой из них создается пункт списка, при этом номер записи станет значением создаваемого пункта. Если у очередной записи перечня номер совпадает с номером, записанным в поле записи вторичной таблицы, этот пункт делается изначально выбранным.

  Код, создающий подобный список, показан в листинге 32.8. Обратите особое внимание на фрагмент, делающий пункт списка изначально выбранным.

Листинг 32.8

```
<p>Категория:<br>
<select name="category">
  @foreach ($cats as $cat)
    <option value="{{ $cat->id }}" @if (old("category",
    $subcat->category) == $cat->id) selected
    @endif>{{ $cat->name }}</option>
  @endforeach
</select></p>
```

Для сравнения используется либо занесенный ранее номер категории (если страница выводится повторно после ошибки ввода), либо номер, сохраненный в подкатегории (если страница выводится впервые), и сравнивается с номером очередной категории. Если эти значения равны, в тег `<option>`, создающий пункт списка, вставляется атрибут `selected`, делающий этот пункт изначально выбранным.

Для сохранения записи вторичной таблицы в этом случае можно применить приемы, описанные ранее. Единственное: в классе модели следует сделать поле, хранящее связанную запись, доступным для массового присваивания (так, в случае подкатегории — это поле, хранящее номер категории).

Связь "многие-ко-многим"

Если создаваемая запись таблицы связана с записями другой таблицы связью «многие-ко-многим», дело несколько усложняется.

В *главе 28* рассматривался пример: таблица стилей (`articles`), связанная такого рода связью с таблицей тегов (`tags`). В этом случае в форме статьи нам следует вывести список тегов, предусмотреть возможность выбора в нем сразу нескольких пунктов и выполнить корректное сохранение тегов, которые выбрал посетитель.

- Сначала, опять же, нужно подготовить перечень записей другой таблицы, связанной с текущей (перечень 1), и передать его шаблону с формой в составе контекста данных. Пример:

  ```
  $ttags = Tag::get();
  return view("articles.input", ["article" => $article,
  "ttags" => $ttags]);
  ```

- В коде шаблона следует сформировать массив, хранящий номера записей другой таблицы, уже связанных с записью текущей таблицы (перечень 2). Для этого достаточно:
  - вызвать у объекта класса модели, представляющего создаваемую или исправляемую запись, метод, чье имя совпадает с методом, который устанавливает связь в классе модели (см. *главу 28*);
  - у возвращенной этим методом коллекции записей вызвать описанный в *главе 30* метод `pluck` в формате с одним аргументом, передав в качестве этого аргумента имя поля, хранящего номер записи (`id`). Обязательно нужно отметить, что в этом случае метод `pluck` вернет не массив, а коллекцию номеров связанных записей;
  - у полученной на предыдущем шаге коллекции вызвать не принимающий аргументов метод `toArray`. Он преобразует коллекцию в обычный массив, который и вернет в качестве результата.
- Написать код, создающий список связанных записей. Это выполняется таким же образом, как и в случае связи «один-ко-многим», за двумя исключениями:
  - к имени элемента управления нужно добавить пустые квадратные скобки. Так мы укажем, что этот параметр хранит массив значений;
  - для каждой из записей перечня 1 нужно проверить, содержится ли ее номер в перечне 2. Для этого можно применить функцию `in_array`:

    `in_array(<искомое значение>, <массив>)`

    Если *искомое значение* имеется в *массиве*, функция вернет `true`, в противном случае — `false`.

  Итак, если номер какой-либо записи перечня 1 содержится в перечне 2, то эта запись другой таблицы связана с текущей записью, и представляющий ее элемент списка нужно сделать изначально выбранным.

  Листинг 32.9 содержит код, создающий список тегов, который выводится в составе формы, предназначенной для создания и правки статьи.

Листинг 32.9

```
<?php $ts = old("tags", $article->tags()->pluck("id")->toArray()); ?>
<p>Теги:<br>
<select name="tags[]" multiple>
  @foreach ($ttags as $ttag)
    <option value="{{ $ttag->id }}" @if (in_array($ttag->id, $ts))
    selected @endif>{{ $ttag->name }}</option>
  @endforeach
</select></p>
```

Теперь рассмотрим вопрос сохранения записи. Значения обычных полей сохраняются с применением описанных ранее средств. А для сохранения списка связанных записей следует применить другие инструменты:

- у объекта класса модели, представляющего создаваемую или исправляемую запись, вызывается метод, чье имя совпадает с методом, который устанавливает связь в классе модели;
- у возвращенной этим методом коллекции связанных записей вызывается метод `sync`:

  <коллекция связанных записей>`->sync(`*<номера связываемых записей>*`)`

  Номера связываемых записей указываются в виде массива. Этот массив можно получить из параметра, чье имя совпадает с именем списка, созданного ранее и перечисляющего записи связанной таблицы.

Листинг 32.10 показывает код действия, выполняющего сохранение новой или существующей статьи.

Листинг 32.10

```
public function save(Request $request) {
  if ($request->has("id")) {
    $article = Article::findOrFail($request->id);
    $article->fill($request->all())->save();
  } else {
    $article = Article::create($request->all());
  }
  $article->tags()->sync($request->get("tags"));
  return redirect()->action("ArticleController@index");
}
```

Дополнительные инструменты для создания и правки отдельных записей

Осталось рассмотреть дополнительные инструменты для создания и правки отдельных записей, которые могут быть нам полезны.

Поиск или создание записей

Метод `findOrNew` модели ищет записи по их номерам и, если не находит, создает новую «пустую» запись:

<класс модели>`::findOrNew(`*<номер записи>*`|`*<список номеров записей>*`,`
<список извлекаемых полей>`)`

Аргументы этого метода указываются в том же виде, что и у метода `find` (см. *главу 30*). В качестве результата возвращается либо найденная запись, либо коллекция найденных записей, либо «пустая» запись.

Пример:

```
$cat = Category::findOrNew(20);
```

Метод `firstOrNew` модели ищет первую запись, удовлетворяющую заданным критериям, и, если не находит подходящей, создает новую «пустую» запись:

```
<класс модели>::firstOrNew(<критерии поиска>)
```

Критерии поиска задаются в виде ассоциативного массива. Ключи его элементов обозначают поля таблицы, а значения элементов — значения данных полей, по которым будет выполняться поиск. В качестве результата возвращается либо найденная запись, либо созданная «пустая» запись.

В следующем примере мы ищем категорию с названием `"Web"` и, если таковая отсутствует, создаем ее:

```
$cat = Category::firstOrNew(["name" => "Web"]);
if (!($cat->id)) {
  $cat->name = "Web";
  $cat->save();
}
```

Метод `firstOrCreate` модели ищет первую запись, удовлетворяющую заданным критериям, и, если не находит подходящей, создает и сохраняет новую запись:

```
<класс модели>::firstOrCreate(<критерии поиска>[, <значения полей>])
```

Критерии поиска задаются в таком же формате, что и у метода `firstOrNew`. *Значения полей*, которые будут занесены в создаваемую запись, указываются в виде ассоциативного массива, где ключи элементов станут именами полей, а значения элементов — значениями этих полей. Если *значения полей* не заданы, в их качестве будут использованы *критерии поиска*. В качестве результата возвращается либо найденная, либо созданная запись.

В следующем примере мы выполняем те же действия, что и в предыдущем примере, но значительно меньшим кодом:

```
$cat = Category::firstOrCreate(["name" => "Web"]);
```

Исправление или создание записей

Метод `updateOrCreate` модели может быть весьма полезен. Он ищет запись по указанным критериям и либо исправляет ее, либо, если не находит подходящей, создает и сохраняет новую запись:

```
<класс модели>::updateOrCreate(<критерии поиска>[, <значения полей>])
```

Критерии поиска и *значения полей* задаются в таком же формате, что и у метода `firstOrCreate`. Второй аргумент указывает значения, которые требуется изменить. В качестве результата возвращается либо найденная и исправленная, либо созданная запись.

В следующем примере мы ищем категорию с названием `"Web"` и меняем ее название на `"WWW"`. Если такая категория отсутствует, она будет создана:

```
$cat = Category::updateOrCreate(["name" => "Web"], ["name" => "WWW"]);
```

Работа со связанными записями

Целый ряд методов позволят нам работать со связанными записями: выполнять их связывание и разрывать связь.

Скажем сразу, что все эти методы следует вызывать у объекта класса, представляющего связь. Этот объект можно получить, вызвав метод, чье имя совпадает с методом, который устанавливает связь в классе модели.

Метод `save` добавляет запись вторичной модели в число связанных с записью первичной модели, у которой он был вызван, при необходимости также сохраняя добавляемую запись:

```
<связь>->save(<связываемая запись>)
```

Связываемая запись должна быть представлена объектом класса соответствующей модели. В качестве результата будет возвращена либо связываемая запись, либо `false`, если создать связь не удалось.

В следующем примере мы ищем категорию № 9, создаем подкатегорию `"Администрирование"` и добавляем к этой категории в качестве связанной записи. Отметим, что явно сохранять новую подкатегорию необязательно — это сделает метод `save`:

```
$cat = Category::find(9);
$subcat = new Subcategory(["name" => "Администрирование",
"description" => "Администрирование операционных систем"]);
$cat->subcategories()->save($subcat);
```

Метод `saveMany` добавляет сразу несколько записей вторичной модели в число связанных с текущей записью первичной модели:

```
<связь>->saveMany(<список связываемых записей>)
```

Список связываемых записей указывается в виде массива. Этот же массив возвращается в качестве результата.

В следующем примере мы добавляем в категорию № 11 сразу две новые подкатегории:

```
$cat = Category::find(11);
$subcats = [
  new Subcategory(["name" => "Шутеры",
  "description" => "Статьи о шутерах"]),
  new Subcategory(["name" => "Стратегии",
  "description" => "Статьи о стратегиях"])
];
$cat->subcategories()->saveMany($subcats);
```

Метод `create` создает и сохраняет запись вторичной модели на основе переданных ему данных и добавляет ее в число связанных с текущей записью первичной модели:

```
<связь>->create(<значения полей>)
```

Значения полей указываются в том же формате, что и у метода firstOrCreate. В качестве результата возвращается созданная запись. Пример:

```
$cat = Category::find(9);
$cat->subcategories()->create(["name" => "Администрирование",
"description" => "Администрирование операционных систем"]);
```

Метод createMany выполняет эту операцию сразу над несколькими записями:

```
<связь>->createMany(<список создаваемых записей>)
```

Список создаваемых записей указывается в виде массива, каждый элемент которого представляет собой массив значений полей для одной из создаваемых записей. В качестве результата возвращается массив созданных записей. Пример:

```
$cat = Category::find(11);
$subcats = [
  ["name" => "Шутеры", "description" => "Статьи о шутерах"],
  ["name" => "Стратегии", "description" => "Статьи о стратегиях"]
];
$cat->subcategories()->createMany($subcats);
```

Метод associate связывает указанную запись первичной модели с текущей записью вторичной модели:

```
<связь>->associate(<связываемая запись>)
```

В качестве результата возвращается указанная запись первичной модели.

В следующем примере мы связываем подкатегорию № 2 с категорией № 1:

```
$subcat = Subcategory::find(2);
$cat = Category::find(1);
$subcat->cat()->associate($cat);
$subcat->save();
```

А не принимающий аргументов метод disassociate разрывает созданную ранее связь. В качестве результата он возвращает указанную запись первичной модели.

В следующем примере мы разрываем связь между подкатегорией № 3 и категорией, с которой она была связана ранее:

```
$subcat = Subcategory::find(3);
$subcat->cat()->disassociate();
$subcat->save();
```

Следующие методы применимы только к моделям, связанным с применением связи «многие-ко-многим».

Метод attach связывает указанную запись или записи с текущей:

```
<связь>->attach(<номер связываемой записи>|
<список номеров связываемых записей>)
```

Можно указать либо номер одной записи, либо массив номеров сразу нескольких записей. Результата этот метод не возвращает.

В следующем примере мы связываем со статьей № 5 теги с номерами 1 и 2:

```
$article = Article::find(5);
$article->tags()->attach([1, 2]);
```

Метод `detach` удаляет связь указанной записи или записей с текущей:

```
<связь>->detach(<номер связанной записи>|
<список номеров связанных записей>)
```

В следующем примере мы удаляем связь статьи № 7 с тегом № 3:

```
$article = Article::find(7);
$article->tags()->detach(3);
```

Проверка введенных в форму данных на корректность. Валидаторы

Перед тем как выполнить создание или изменение записи на основе введенных в форму данных, следует проверить эти данные на корректность. Какую-то часть этой проверки мы можем реализовать на стороне клиента, воспользовавшись инструментами валидации, которые поддерживаются DOM (см. *главу 19*). Однако основную проверку следует выполнить все же на стороне сервера.

Для проверки данных Laravel предоставляет инструменты, называемые *валидаторами*. Валидаторы бывают нескольких разновидностей.

Простейшие валидаторы

В самом простом случае валидацию можно выполнить вызовом одного метода. Этот метод задействует простейший валидатор.

Полностью автоматическая валидация

Класс `Controller`, от которого наследуются все контроллеры проекта, использует трейт `ValidatesRequests`. Этот трейт поддерживает метод `validate`, который и выполняет валидацию:

```
<контроллер>->validate(<запрос>, <список правил валидации>
[, <список сообщений об ошибках>])
```

`Запрос` представляется объектом класса `Request`. Его можно получить из параметра, принимаемого методом-действием.

`Список правил валидации` указывается в виде ассоциативного массива. Ключи элементов этого массива должны совпадать с именами GET- и POST-параметров, через которые передаются данные (имена этих параметров, как мы помним, — суть имена элементов управления в форме), а значения элементов задают сами правила валидации, записанные в виде строк (мы рассмотрим их позже).

Список сообщений об ошибках также записывается в виде ассоциативного массива. Ключи его элементов указывают на GET- или POST-параметр, а также на одно из правил валидации, при нарушении которого должно быть выведено это сообщение, и задаются в формате `<имя параметра>.<правило валидации>`. Значения же элементов должны представлять собой сами сообщения об ошибках.

Если *список сообщений об ошибках* не указан, будут выводиться встроенные сообщения об ошибках на английском языке.

Если данные не проходят валидацию, метод `validate` автоматически выполняет перенаправление на предыдущую страницу — то есть на страницу с формой, в которую были занесены эти данные. При этом шаблону передается список сообщений об ошибках (доступен через переменную `errors`) и сами введенные данные (доступны посредством функции `old`). Весь код, следующий за вызовом этого метода, не будет выполнен.

Метод `validate` должен вызываться в теле метода-действия перед любым кодом, выполняющим сохранение записи.

В следующем примере мы проверяем, занесено ли название категории (правило `required` указывает, что в элемент управления обязательно должно быть занесено какое-либо значение):

```
public function save(Request $request) {
  $this->validate($request, ["name" => "required"],
  ["name.required" => "Введите название категории"]);
  // Код, выполняющий сохранение записи
}
```

Полуавтоматическая валидация

При автоматической валидации проходят проверку все данные, полученные от формы в составе запроса. Если же нужно проверить не все данные, а только их часть, или же реализовать какую-либо специфическую логику обработки ошибок ввода, уместнее задействовать полуавтоматическую валидацию.

Сначала нужно создать валидатор. Для этого предназначен метод `make` фасада `Validator`:

```
Validator::make(<проверяемые данные>, <список правил валидации>
[, <список сообщений об ошибках>])
```

Проверяемые данные указываются в виде ассоциативного массива, ключи элементов которого хранят имена полученных GET- и POST-параметров, а значения элементов представляют значения этих параметров. Этот массив можно получить вызовом методов `all`, `only` или `except` запроса (за подробностями — к *главе 30*).

Остальные аргументы этого метода указываются в том же формате, что и аналогичные аргументы метода `validate`.

В качестве результата метод `make` возвращает объект класса `Validator`. Его-то мы и применим для проверки данных. Пример:

```
use Illuminate\Support\Facades\Validator;
. . .
$validator = Validator::make($request->all(), ["name" => "required"],
["name.required" => "Введите название категории"]);
```

Проще всего проверить данные вызовом метода `validate` класса `Validate`. Этот метод не принимает аргументов, не возвращает результата и, если данные не прошли проверку, перенаправляет посетителя на предыдущую страницу вместе с данными и сообщениями об ошибках, — как и одноименный метод контроллера. Пример:

```
$validator->validate();
```

В более сложных случаях нам может понадобиться выполнить все действия по проверке данных и перенаправлению на предыдущую страницу самостоятельно. Здесь нам могут помочь два не принимающих аргументов метода класса `Validate`:

- `passes` — возвращает `true`, если данные корректны, и `false` в противном случае;
- `fails` — возвращает `true`, если данные некорректны, и `false` в противном случае.

Чтобы выполнить перенаправление с отсылкой введенных данных, нужно у объекта класса `RedirectResponse`, возвращенного вызовом функции `redirect` с указанием аргумента, вызвать метод `withInput`:

```
<перенаправление>->withInput([<данные>])
```

Указанные в вызове этого метода *данные* — ассоциативный массив того же формата, который используется в методе `make`, — будут переданы шаблону, на который выполняется перенаправление. Если аргумент не указан, будут переданы данные, взятые из запроса.

Чтобы отправить шаблону еще и сообщения об ошибках, нужно у объекта класса `RedirectResponse`, возвращенного вызовом функции `redirect` с указанием аргумента, вызвать метод `withErrors`:

```
<перенаправление>->withErrors(<валидатор>|<массив ошибок>)
```

В качестве аргумента может быть указан как объект валидатора, так и ассоциативный массив ошибок, в котором ключи выступают именами элементов управления, а значениями являются тексты соответствующих им сообщений об ошибках.

Оба этих метода в качестве результата возвращают тот же объект класса `RedirectResponse`, у которого он был вызван. Пример:

```
if ($validator->passes()) {
  Category::create($request->all());
} else {
  return redirect("categories/create")->withErrors($validator)
  ->withInput();
}
```

Условные правила валидации

Часто бывает нужно применить те или иные правила валидации только, если выполняется какое-либо условие. И в этом случае Laravel приходит нам на помощь.

Класс `Validator`, чей объект возвращается в качестве результата методом `make`, поддерживает метод `sometimes`:

```
<валидатор>->sometimes(<имя параметра>, <правила валидации>,
<функция, проверяющая условие>)
```

Функция, проверяющая условие, должна принимать единственный параметр — полученный запрос — и возвращать в качестве результата логическую величину. Если она вернет `true`, к параметру (элементу управления в форме) с указанным *именем* будут применены *правила валидации*, если же она вернет `false`, *правила* применены не будут.

В следующем примере, если количество созданных категорий превышает 10, к элементу управления `order` (порядок) будет применено правило, требующее, чтобы занесенное в этот элемент числовое значение было не менее 1:

```
$validator-sometimes("order", "min:1", function ($input) {
  return Category::count() > 10;
});
```

Правила валидации

Laravel поддерживает очень много правил валидации. Самые полезные из них приведены далее:

- `required` — в элемент управления должно быть занесено непустое значение;
- `required_if:`*<имя>*`,`*<список значений, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значение элемента управления с указанным *именем* совпадает с одним из перечисленных в *списке*;
- `required_unless:`*<имя>*`,`*<список значений, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значение элемента управления с указанным *именем* не совпадает ни с одним из перечисленных в *списке*;
- `required_with:`*<список имен, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значение было введено в один из элементов, чьи имена перечислены в *списке*;
- `required_with_all:`*<список имен, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значения были введены во все элементы, чьи имена перечислены в *списке*;
- `required_without:`*<список имен, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значение не было введено в один из элементов, чьи имена перечислены в *списке*;

- `required_without_all:`*<список имен, разделенных запятыми>* — в элемент управления должно быть занесено значение, если значения не были введены во все элементы, чьи имена перечислены в *списке*;
- `filled` — в элемент управления, если он присутствует, должно быть занесено значение;
- `present` — в элемент должно быть занесено значение, которое может быть пустой строкой;
- `string` — строка;
- `alpha` — допускаются только буквы;
- `alpha_dash` — допускаются только буквы, дефисы и символы подчеркивания;
- `alpha_num` — допускаются только буквы и цифры;
- `numeric` — число;
- `integer` — целое число;
- `min:`*<значение>* — строка длиной не меньше указанного *значения*, или число, не меньшее указанного *значения*, или файл размером не меньше указанного *значения*, или массив файлов с количеством элементов не меньше указанного *значения*;
- `max:`*<значение>* — строка длиной не больше указанного *значения*, или число, не большее указанного *значения*, или файл размером не больше указанного *значения*, или массив файлов с количеством элементов не больше указанного *значения*;
- `digits:`*<длина>* — числовое значение с указанной *длиной*;
- `digits_between:<мин.>,<макс.>` — числовое значение с указанной в цифрах длиной, находящейся в диапазоне от `мин.` до `макс.`;
- `between:`*<мин.>*`,`*<макс.>* — числовое значение, находящееся в диапазоне от *мин.* до *макс.*;
- `date` — значение даты;
- `date_format:`*<формат>* — значение даты, совпадающее с указанным *форматом*;
- `after:`*<дата>* — значение типа даты или даты и времени, более позднее, чем указанная *дата*;
- `before:`*<дата>* — значение типа даты или даты и времени, более раннее, чем указанная *дата*;

> ***На заметку***
>
> Описание форматов значений даты, поддерживаемых PHP, можно найти по интернет-адресу **http://fi2.php.net/manual/ru/datetime.formats.date.php**.

- `url` — интернет-адрес;
- `ip` — IP-адрес;
- `email` — адрес электронной почты;

- `boolean` — логическое значение. Допускаются значения `true`, `false`, "1", 1 (они преобразуются в `true`), "0" и 0 (они преобразуются в `false`);
- `regex:`*<регулярное выражение>* — значение должно совпадать с *регулярным выражением*;
- `in:`*<список значений, разделенных запятыми>* — значение должно совпадать с одним из указанных в *списке*;
- `not_in:`*<список значений, разделенных запятыми>* — значение не должно совпадать ни с одним из указанных в *списке*;
- `same:`*<имя>* — значение должно совпадать с тем, что занесено в элемент управления с указанным *именем*;
- `different:`*<имя>* — значение должно отличаться от того, что занесено в элемент управления с указанным *именем*;
- `unique:`*<таблица>*`[,`*<поле>*`]` — значение должно быть уникальным среди всех хранящихся в указанном *поле* указанной *таблицы*. Если *поле* не задано, будет использовано поле с именем, совпадающим с именем элемента управления;
- `exists:`*<таблица>*`[,`*<поле>*`]` — значение должно совпадать с одним из хранящихся в указанном *поле* указанной *таблицы*. Если *поле* не задано, будет использовано поле с именем, совпадающим с именем элемента управления;
- `file` — файл любого типа;
- `mimetypes:`*<список MIME-типов, разделенных запятыми>* — файл, чей MIME-тип совпадает с одним из перечисленных в *списке*;
- `mimes:`*<список расширений, разделенных запятыми>* — файл с расширением, совпадающим с одним из перечисленных в *списке* (расширения указываются без начальных точек);
- `image` — графический файл формата GIF, JPEG, PNG, SVG или BMP;
- `dimensions:`*<параметры>* — графический файл, хранящий изображение с указанными *параметрами*. В качестве параметров можно указать следующие:
  - `min_width=`*<значение>* — ширина не меньше указанного *значения*;
  - `max_width=`*<значение>* — ширина не больше указанного *значения*;
  - `min_height=`*<значение>* — высота не меньше указанного *значения*;
  - `max_height=`*<значение>* — высота не больше указанного *значения*;
  - `width=`*<значение>* — ширина, строго равная указанному *значению*;
  - `height=`*<значение>* — высота, строго равная указанному *значению*;
  - `ratio=`*<значение>* — соотношение сторон, строго равное указанному *значению*. В качестве такового допускаются числа с плавающей точкой (`1.33`) и дроби (`16/9`);

- `accepted` — должно храниться значение `"yes"`, `"on"`, `1` или `true`. Обычно применяется к флажкам, установка которых означает согласие с некоторыми условиями;
- `confirmed` — значение должно совпадать со значением, занесенным в элемент управления с именем вида `<имя>_confirmation` (например, значение поля ввода `password` должно совпадать со значением, занесенным в поле `password_confirmation`);
- `nullable` — элемент управления может содержать значение `null`;
- `sometimes` — следующие правила будут применены только в том случае, если этот элемент управления присутствует в форме;
- `bail` — прервать валидацию, если какое-либо правило из указанных далее не выполняется (по умолчанию в этом случае проверяются все последующие правила).

Можно объединять правила, применяемые к одному элементу управления, разделяя их символами вертикальной черты.

Вот несколько примеров:

- значение в элементе управления `name` должно быть обязательно указано:

  ```
  "name" => "required"
  ```

- значение в элементе управления `name` должно быть обязательно указано, должно включать не более 50 символов и быть уникальным среди значений, хранящихся в поле `name` таблицы `categories`:

  ```
  "name" => "required|max:50|unique:categories,name"
  ```

- значение в элементе управления `picture` должно представлять собой графический файл размером не более 16 Кбайт, хранящий изображение с максимальной шириной 400 пикселов, максимальной высотой 300 пикселов и соотношением сторон 4/3:

  ```
  "picture" => "image|max:16|dimensions:max_width=400,max_height=300,↳
  ratio=4/3"
  ```

Запросы форм

И наконец, есть способ максимально автоматизировать проверку данных на корректность, полностью переложив ее и все сопутствующие действия на плечи Laravel. Однако для этого придется написать довольно много кода.

Прежде всего, следует объявить особый класс, представляющий *запрос формы*, — запрос, предназначенный для пересылки данных, которые введены в одну конкретную форму и предназначены для сохранения в одной конкретной модели. В этом классе мы опишем правила валидации и сообщения, которые будут выведены в случае ошибок ввода.

Написать класс запроса формы нам помогут средства прототипирования Laravel. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду вида:

```
php artisan make:request <имя класса запроса>
```

Все файлы с классами запросов форм сохраняются в папке App\Http\Requests. Эти классы находятся в пространстве имен `App\Http\Requests` и порождаются от класса `FormRequest`.

Код только что сгенерированного класса запроса формы `CategoryRequest`, предназначенного для валидации категорий, показан в листинге 32.11.

Листинг 32.11

```
<?php
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

class CategoryRequest extends FormRequest
{
  public function authorize()
  {
    return false;
  }

  public function rules()
  {
    return [
      //
    ];
  }
}
```

Первое, что нам следует сделать, — указать в качестве значения, возвращаемого методом `authorize` этого класса, логическую величину `true`. Если мы этого не сделаем, Laravel заблокирует пересылку данных от формы действию. (Более подробно о методе `authorize` и его назначении мы поговорим в *главе 33*.)

Теперь поработаем над кодом публичного метода `rules`. В качестве результата он должен вернуть ассоциативный массив правил валидации, формируемый по тем же принципам, что и аналогичный массив, передаваемый вторым аргументом методу `validate` контроллера (см. ранее).

Для указания сообщений об ошибках следует дополнительно объявить в классе запроса формы публичный метод `messages`. Он не должен принимать параметров и в качестве результата должен возвращать ассоциативный массив, аналогичный тому, что передается третьим аргументом методу `validate` контроллера.

Листинг 32.12 показывает полный код класса `CategoryRequest`, выполняющего проверку, было ли указано название категории, и задающий соответствующее сообщение об ошибке.

Листинг 32.12

```
<?php
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

class CategoryRequest extends FormRequest {
  public function authorize() {
    return true;
  }

  public function rules() {
    return ["name" => "required"];
  }

  public function messages() {
    return [
      "name.required" => "Название категории должно быть указано"
    ];
  }
}
```

После этого нам следует в объявлении метода-действия контроллера, выполняющего получение данных из формы, указать в качестве типа параметра запроса только что созданный нами класс. Пример:

```
use App\Http\Requests\CategoryRequest;
. . .
public function save(CategoryRequest $request) { . . . }
```

При этом код, выполняющий валидацию, из тела метода-действия можно удалить — все, что нужно, сделает запрос формы.

Массовые создание, правка и удаление записей

Далеко не всегда приходится создавать, исправлять и удалять по одной записи за один раз. Может понадобится создать сразу несколько записей, изменить значение какого-либо поля во всех записях, удовлетворяющих определенному критерию, или даже удалить все такие записи.

Все описанные здесь методы поддерживаются классом `Builder`. Они вызываются так же, как и методы, создающие запросы к базе данных (см. *главу 30*).

Метод `insert` выполняет создание записей:

```
insert(<значения полей>|<список создаваемых записей>)
```

Можно создать как одну запись, передав в качестве аргумента ассоциативный массив со *значениями полей*, так и несколько записей, передав им *список записей* — массив ассоциативных массивов, хранящих значения полей. В качестве результата возвращается `true`, если добавление записей было успешно выполнено, и `false` в противном случае. Пример:

```
Category::insert([
  "name" => "Защита",
  "description" => "Статьи о компьютерной безопасности",
  "order" => 2
]);
```

Также может оказаться полезным метод `insertGetId`, который создает запись и возвращает ее номер:

```
insertGetId(<значения полей>)
```

Пример:

```
$newCatId = Category::insertGetId([
  "name" => "Защита",
  "description" => "Статьи о компьютерной безопасности",
  "order" => 2
]);
```

Метод `update` выполняет массовое исправление записей:

```
update(<список новых значений>)
```

В качестве результата он возвращает количество исправленных записей.

В следующем примере мы у всех статей, созданных перед 2016 годом, заносим в поле `published` значение `false`:

```
Article::whereYear("created_at", "<", 2016)
->update(["published" => false]);
```

Метод `increment` увеличивает значение заданного поля на указанную величину:

```
increment(<имя поля>[, <величина>[, <список новых значений>]])
```

Если *величина* не указана, она принимается равной 1. Также можно заодно изменить значения других полей, указав их в *списке новых значений*. В качестве результата этот метод возвращает количество исправленных записей.

Вот пара примеров:

- у всех статей, созданных 31 декабря 2016 года, увеличиваем на единицу значение поля `rating`:

  ```
  Article::whereDate("created_at", "2016-12-31")->increment("rating");
  ```

- а у всех статей, созданных 1 января 2017 года, увеличиваем на `5` значение поля `rating` и дополнительно заносим в поле `order` число `10`:

  ```
  Article::whereDate("created_at", "2017-01-01")
  ->increment("rating", 5, ["order" => 10]);
  ```

Метод `decrement`, напротив, уменьшает значение заданного поля на указанную величину. Он вызывается так же, как метод `increment`, и возвращает аналогичный результат.

В следующем примере мы у всех статей, созданных 31 декабря 2015 года или раньше, уменьшаем значение поля `rating` на `10`:

```
Article::whereDate("created_at", "<=" "2015-12-31")
->decrement("rating", 10);
```

Не принимающий аргументов метод `delete` удаляет записи и возвращает в качестве результата количество удаленных записей.

В следующем примере мы удаляем все статьи, созданные перед 2010 годом:

```
Article::whereYear("created_at", "<", 2010)->delete();
```

Не принимающий аргументов и не возвращающий результатов метод `truncate` удаляет все записи в таблице:

```
Comment::truncate();
```

Работа с выгруженными файлами

Последнее, что мы рассмотрим в этой главе, — сохранение выгруженных посетителем файлов и работу с ними.

> ***Внимание!***
>
> Здесь будет рассмотрено только сохранение выгруженных файлов на локальном диске серверного компьютера. (Laravel также поддерживает сохранение файлов в облачных хранилищах — как это выполнить, описано в документации по фреймворку.)

Файловое хранилище и диски Laravel

Перед тем как реализовывать сохранение выгруженных файлов, следует настроить подсистему, выполняющую эту задачу, — *хранилище*. Мы будем использовать локальное хранилище, сохраняющее файлы на локальном диске.

Локальное хранилище может использовать два местоположения для сохранения файлов, или, в терминологии Laravel, *диска*:

- локальный — сохраняет файлы в папке storage\app (используется по умолчанию);
- публичный — сохраняет файлы в папке storage\app\public.

Папка storage и все вложенные в нее папки недоступны Web-серверу. Поэтому, чтобы дать посетителю возможность загружать хранящиеся в них файлы, нам придется создавать действие контроллера и применять в его коде метод `download` или `file` (они были описаны в *главе 30*).

Разработчики Laravel рекомендуют применять публичный диск для хранения файлов, которые следует сделать доступными для всех, а локальный диск — для хра-

нения частных файлов отдельных посетителей. Хотя, в принципе, не имеет особого значения, какой из дисков использовать.

Настройки файлового хранилища и дисков хранятся в конфигурационном файле filesystems.php, размещенном в папке config папки проекта. Мы можем изменить диск, на котором выполняется сохранение файлов по умолчанию, и пути к папкам дисков.

Используемый по умолчанию диск указывается в следующей строке, в параметре `default`:

```
'default' => 'local',
```

Значение `'local'` предписывает использовать локальный диск, а `'public'` — публичный.

Настройки дисков записываются в параметре `disks`. Соответствующий фрагмент кода файла config\filesystem.php показан в листинге 32.13.

Листинг 32.13

```
'disks' => [
  'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
  ],

  'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'visibility' => 'public',
  ],
  . . .
],
```

Разделы `local` и `public` задают параметры, соответственно, локального и публичного диска.

Здесь нас более всего интересует параметр `root`, который указывает путь к папке, соответствующей диску. Отметим, что для задания путей применяются функции, возвращающие пути к папкам проекта и описанные в *главе 30*.

Если мы укажем для хранения файлов папки, вложенные в папку public, то отпадет потребность в написании отдельного действия, выполняющего отправку файлов. Тогда, чтобы вывести файл на экран или создать на него гиперссылку, нам достаточно будет лишь написать нужный тег с интернет-адресом, указывающим на этот файл.

Особенности создания Web-формы для выгрузки файлов

При создании формы, из которой будет выгружаться файл, следует указать для нее метод кодирования данных `multipart/form-data` (он, как мы помним, задается атрибутом `enctype` тега `<form>`). Если этого не сделать, файлы не будут выгружены. Пример:

```
<form . . . enctype="multipart/form-data">
  . . .
  <p>Файл:<br>
  <input type="file" name="file"></p>
  . . .
</form>
```

Если планируется использовать поле ввода файла с возможностью указания сразу нескольких файлов, к имени этого поля нужно добавить квадратные скобки. Так мы укажем, что значение этого элемента управления представляет собой массив. Пример:

```
<input type="file" name="files[]" multiple>
```

Получение и сохранение выгруженных файлов

Обработка выгруженных посетителем файлов выполняется либо в действии контроллера, либо в обработчике события модели (о которых говорилось в *главе 28*).

Получение выгруженного файла и сведений о нем

Для получения выгруженного файла следует вызвать метод `file`, поддерживаемый классом `Request`, то есть запросом:

```
<запрос>->file(<имя параметра>)
```

Если параметр с указанным *именем* хранит файл, будет возвращен представляющий его объект класса `UploadedFile`. Если этот параметр хранит сразу несколько выгруженных файлов, будет возвращен массив объектов этого класса. Если же заданный параметр отсутствует, будет возвращено значение `null`. Пример:

```
public function save(Request $request) {
  $uploaded = $request->file("file");
  . . .
}
```

Не принимающий аргументов метод `allFiles` запроса возвращает массив всех выгруженных файлов.

Метод `hasFile` запроса позволяет проверить, был ли выгружен файл:

```
<запрос>->hasFile(<имя параметра>)
```

Если параметр с указанным *именем* существует и хранит файл, будет возвращено значение `true`, в противном случае — `false`. Пример:

```
if ($request->hasFile("file")) {
  // Файл был выгружен. Обрабатываем его
}
```

Объект `UploadedFile` поддерживает ряд методов, позволяющих выяснить параметры выгруженного файла. Эти методы приведены далее, и все они не принимают аргументов:

- `getClientOriginalName` — возвращает имя файла с расширением;
- `getClientOriginalExtension` — возвращает расширение без начального символа точки;
- `getMimeType` — возвращает MIME-тип файла;
- `getSize` — возвращает размер файла в виде числа в байтах.

Пример:

```
$fileName = $uploaded->getClientOriginalName();
```

Сохранение выгруженного файла

Для сохранения выгруженного файла применяются следующие два метода класса `UploadedFile`.

Метод `store` сохраняет файл под именем, сгенерированном самим Laravel:

```
<файл>->store(<папка>[, <диск>]);
```

Первым аргументом указывается *папка*, в которую будет сохранен файл. Эта папка должна быть вложена в папку, соответствующую выбранному диску. Если требуется поместить файл непосредственно в папку диска, следует указать первым аргументом пустую строку.

Вторым аргументом может быть указано обозначение *диска*, в который будет помещен выгруженный файл. Если он не указан, файл будет сохранен на диске, используемом по умолчанию.

Файл всегда сохраняется под именем, сгенерированным самим Laravel. Это имя формируется на основе содержимого файла. Нужно иметь в виду, что, если файл очень велик, генерирование имени может занять продолжительное время.

В качестве результата метод `store` возвращает строку с полным путем к сохраненному файлу или `false`, если сохранить файл не удалось.

Вот несколько примеров:

- сохраняем полученный файл в папке uploads, находящейся в папке, что соответствует диску. Поскольку второй аргумент не указан, будет использован диск по умолчанию (локальный). В результате файл будет сохранен в папке storage\app\uploads под автоматически сгенерированным именем:

  ```
  $uploaded->store("uploads");
  ```

- сохраняем в той же папке несколько выгруженных файлов:

```
$uploads = $request->file("files");
foreach ($uploads as $uploaded) {
  $uploaded->save("uploads");
}
```

- сохраняем файл непосредственно в папке, соответствующей публичному диску. В результате файл сохранится в папке storage\app\public:

```
$uploaded->store("", "public");
```

Аналогичный метод `storeAs` сохраняет файл под указанным именем:

```
<файл>->storeAs(<папка>, <имя сохраняемого файла>[, <диск>]);
```

Пример:

```
$uploaded->storeAs("uploads", "uploaded_file.dat");
```

> ***Внимание!***
>
> При задании имени сохраняемого файла следует иметь в виду, что PHP некорректно обрабатывает символы кириллицы, присутствующие в именах файлов, преобразуя их в нечитаемые последовательности символов. Поэтому следует исключить применение в именах файлов кириллических букв.

Работа с выгруженными файлами

Ряд методов, поддерживаемых фасадом `Storage`, выполняет различные манипуляции с файлами, из которых нас, в первую очередь, интересует проверка существования файла и его удаление.

По умолчанию все эти методы работают с файлами, хранящимися на текущем диске. (Однако мы можем указать другой диск — как это сделать, будет описано далее.) Пути к файлам, задаваемые в качестве их аргументов, указываются относительно папки, соответствующей используемому диску.

Метод `exists` позволяет проверить существование файла:

```
exists(<путь к файлу>)
```

Метод возвращает `true`, если файл существует, и `false` в противном случае.

В следующем примере мы проверяем, существует ли файл file.zip, хранящийся в папке uploads папки, которая соответствует текущему (по умолчанию — локальному) диску:

```
use Illuminate\Support\Facades\Storage;
. . .
if (Storage::exists("uploads\file.zip")) {
  // Файл существует
}
```

Метод `delete` удаляет один или сразу несколько файлов:

```
delete(<путь к файлу>|<список путей к файлам>)
```

Можно указать либо *путь* к одному удаляемому файлу, либо целый их *список*, заданный в виде массива. Метод возвращает `true`, если удаление было выполнено успешно, и `false` в противном случае.

Вот пара примеров:

- удаляем файл file1.zip из папки uploads:

  ```
  Storage::delete("uploads\file1.zip");
  ```

- удаляем файл file2.zip из папки uploads и file3.exe из папки private:

  ```
  Storage::delete(["uploads\file2.zip", "private\file3.exe"]);
  ```

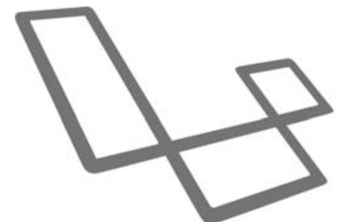
Если предстоит работать с файлами, расположенными на другом диске, отличном от указанного по умолчанию в настройках, нужно предварительно указать этот диск в вызове метода `disk`:

```
disk(<обозначение диска>)
```

Этот метод следует вызвать перед вызовом других методов, а последние вызывать у возвращенного им результата.

В следующем примере мы удаляем файл file1.zip из папки uploads публичного диска:

```
Storage::disk("public")->delete("uploads\file1.zip");
```



ГЛАВА 33

Разграничение доступа. Использование CAPTCHA

Разрешать править внутренние данные нашего сайта — списки категорий, подкатегорий, статей и комментариев — следует далеко не всем. Нам придется реализовать систему разграничения доступа, которая позволит добраться до этих данных только тем посетителям, которые, во-первых, внесены в список пользователей, во-вторых, выполнили процедуру входа и, в-третьих, имеют достаточные права для этого. Этим-то мы сейчас и займемся.

Встроенная подсистема разграничения доступа Laravel

Laravel изначально поставляется с довольно развитой подсистемой разграничения доступа. Она имеет следующие особенности:

- любой посетитель может зарегистрироваться на сайте, перейдя на страницу с Web-формой регистрации и занеся в нее все необходимые данные, — в первую очередь, адрес электронной почты и пароль;
- зарегистрированный пользователь может выполнить вход на сайт, перейдя на страницу с формой входа и введя в нее адрес электронной почты и пароль, указанные при регистрации;
- пользователь, выполнивший процедуру входа, может выйти с сайта, также перейдя по определенному интернет-адресу;
- забывший пароль пользователь может перейти на страницу с формой восстановления пароля и указать в ней заданный при регистрации адрес электронной почты. На этот адрес ему будет выслано письмо с гиперссылкой, перейдя по которой, он попадет на страницу, где сможет ввести новый пароль вместо забытого;
- при попытке посетителя попасть на страницу с ограниченным доступом будет выполнено его перенаправление на страницу входа;
- если пользователь несколько раз подряд ввел неправильные адрес электронной почты и (или) пароль, то следующую попытку войти на сайт он сможет пред-

принять лишь спустя минуту. Это сделано для предотвращения атак путем подбора пароля.

Эта подсистема включает в себя следующие программные модули:

- набор контроллеров, находящихся в пространстве имен `App\Http\Controllers\Auth` (следовательно, файлы, хранящие их код, располагаются в одноименной папке):
  - `RegisterController` — выполняет регистрацию нового пользователя;
  - `LoginController` — выполняет процедуру входа на сайт и выхода с сайта;
  - `ForgotPasswordController` — служит для запуска процедуры сброса пароля, а именно отправки по введенному посетителем адресу электронной почты письма с гиперссылкой, ведущей на страницу сброса пароля;
  - `ResetPasswordController` — выполняет процедуру сброса пароля;
- модель `User`, представляющую зарегистрированного пользователя;
- две миграции:
  - <*дата и время*>_create_users_table.php — создает таблицу списка пользователей `users`;
  - <*дата и время*>_create_password_resets_table.php — создает таблицу `password_resets`, хранящую идентификаторы запросов на сброс пароля.

Так что нам остается лишь выполнить миграции (как это сделать, было описано в *главе 27*. Впрочем, если какие-либо миграции уже выполнялись ранее, обе упомянутые здесь таблицы уже должны быть созданы), записать маршруты, указывающие на действия контроллеров, и создать шаблоны для соответствующих страниц.

Но даже и в этом Laravel идет нам навстречу. Чтобы создать маршруты и шаблоны, достаточно открыть консоль OpenServer, перейти в папку проекта и отдать команду:

```
php artisan make:auth
```

Она выполнит следующие действия:

- добавит в код маршрутизатора (в файл routes\web.php) выражение:

  ```
  Auth::routes();
  ```

  которое создаст набор маршрутов следующего вида:

  ```
  Route::get('login', 'Auth\LoginController@showLoginForm')
  ->name('login');
  Route::post('login', 'Auth\LoginController@login');
  Route::post('logout', 'Auth\LoginController@logout')->name('logout');
  Route::get('register', 'Auth\RegisterController@showRegistrationForm')
  ->name('register');
  Route::post('register', 'Auth\RegisterController@register');
  Route::get('password/reset',
  'Auth\ForgotPasswordController@showLinkRequestForm');
  Route::post('password/email',
  ```

```
'Auth\ForgotPasswordController@sendResetLinkEmail');
Route::get('password/reset/{token}',
'Auth\ResetPasswordController@showResetForm');
Route::post('password/reset', 'Auth\ResetPasswordController@reset');
```

- создаст в папке resources\views папку auth с шаблонами:
  - register.blade.php — страница с формой регистрации;
  - login.blade.php — страница с формой входа;
  - passwords\email.blade.php — страница с формой, в которую заносится адрес электронной почты для отправки письма с гиперссылкой на страницу сброса пароля;
  - passwords\reset.blade.php — страница сброса пароля;
- создаст в папке resources\views папку layouts с шаблоном app.blade.php, от которого наследуются все упомянутые здесь четыре шаблона.

> **ВНИМАНИЕ!**
> Если в папке *resources\views* уже есть папка *layouts* с шаблоном *app.blade.php*, он будет перезаписан. Поэтому разработчики Laravel рекомендуют выполнять команду создания маршрутов и шаблонов сразу после создания нового проекта.

После всего этого мы можем использовать полностью работоспособную подсистему в своем проекте.

Ограничение доступа к страницам

Подготовив список пользователей, необходимые маршруты, контроллеры и шаблоны, мы можем приступить к реализации собственно ограничения доступа к страницам сайта. Мы укажем, какие действия должны быть доступны лишь для зарегистрированных пользователей, какие — для всех посетителей, включая гостей, а какие — только зарегистрированным пользователям, удовлетворяющим определенным условиям.

Простые случаи ограничения доступа

Проще всего сделать так, чтобы определенные или даже все действия какого-либо контроллера были доступны лишь для зарегистрированных пользователей.

- Чтобы сделать все действия контроллера доступными лишь для зарегистрированных пользователей, нужно указать для контроллера посредник `auth` в конструкторе его класса:

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

Теперь все действия контроллера `UserController` будут доступны лишь для зарегистрированных пользователей.

- Чтобы сделать доступными для зарегистрированных пользователей лишь некоторые действия контроллера, можно пойти тремя путями:
  - указать для нужных действий посредник `auth` непосредственно в соответствующих им маршрутах:

    ```
    Route::get("/categories/create", "CategoryController@create")
    ->middleware("auth");
    ```

    Действие `create` контроллера `CategoryController` отныне будет доступно только для зарегистрированных пользователей;
  - указать посредник `auth` в конструкторе класса контроллера только для нужных действий:

    ```
    class CategoryController extends Controller {
      public function __construct() {
        $this->middleware("auth")->only(["input", "save"]);
      }
      . . .
    }
    ```

    Действия `input` и `save` контроллера `CategoryController` станут доступными только для зарегистрированных пользователей;
  - указать посредник `auth` в конструкторе класса контроллера для всех действий, кроме тех, что должны быть доступны всем:

    ```
    class CategoryController extends Controller {
      public function __construct() {
        $this->middleware("auth")->except("index");
      }
      . . .
    }
    ```

    Действие `index` контроллера `CategoryController` станет доступным всем, остальные действия — только зарегистрированным пользователям.

Ограничение доступа на основе более сложных условий

В большинстве случаев ограничение доступа можно реализовать описанными ранее способами. Но часто бывает необходимо дополнительно ограничить доступ к действиям на основе более сложных условий. Так, автору статей следует предоставить возможность править и удалять только те статьи, которые написал он сам. Как быть в таком случае?

Laravel предусматривает два подхода реализации дополнительного ограничения доступа на основе сложных условий: шлюзы и политики.

Шлюзы

Шлюз — это обычная функция, которая проверяет выполнение условия на основе переданных ей аргументов и возвращает в качестве результата логическую величину: `true`, если операция разрешена, и `false`, если она запрещена.

Шлюзы записываются в коде метода `boot` провайдера `AuthServiceProvider`. Файл с этим классом находится в папке app\Providers, как и все остальные провайдеры проекта.

Шлюз регистрируется вызовом метода `define` фасада `Gate`:

```
Gate::define(<выполняемая операция>, <функция-шлюз>)
```

Первым аргументом указывается выполняемая операция в виде строки. Эта операция может иметь любое название: `"create-category"`, `"update-article"` и т. п.

Вторым аргументом указывается функция, собственно, реализующая шлюз. Она может принимать, по крайней мере, один параметр — объект класса модели, представляющий пользователя, который в данный момент выполнил вход на сайт. Остальными параметрами могут быть объект класса модели, представляющий исправляемую или удаляемую запись, и др. В качестве результата функция должна возвращать `true`, если операция должна быть разрешена, и `false` в противном случае.

Листинг 33.1 показывает фрагмент кода провайдера `AuthServiceProvider`, в котором зарегистрирован шлюз, разрешающий правку и удаление статей только тем пользователям, которые являются их авторами.

Листинг 33.1

```
use Illuminate\Support\Facades\Gate;
class AuthServiceProvider extends ServiceProvider {
  public function boot() {
    Gate::define("update-delete-article", function ($user, $article) {
      return $article->author == $user->id;
    });
  }
}
```

Выполняемая операция получила имя `"update-delete-article"`. Как видим, здесь выполняется проверка равенства номера автора, сохраненного в данных статьи, и номера текущего пользователя, и возвращается результат этого сравнения.

Проверку, может ли пользователь выполнить действие с указанным при регистрации шлюза названием, можно выполнить вызовом одного из двух описанных далее методов фасада `Gate`;

- `Gate::allows(<выполняемая операция>, <аргументы, передаваемые шлюзу>)` — возвращает `true`, если эта *операция* разрешена, и `false` в противном случае. Пример:

```
use Illuminate\Support\Facades\Gate;
. . .
if (Gate::allows("update-delete-article", $article)) {
  // Статью можно исправить или удалить
}
```

- `Gate::denies(`*<выполняемая операция>*`,` *<аргументы, передаваемые шлюзу>*`)` — возвращает `true`, если эта *операция* запрещена, и `false` в противном случае.

Политики

Политика — это класс, поддерживающий набор особых методов. Каждый из этих методов соответствует выполняемой с записями модели операции (просмотр списка, просмотр отдельной записи, добавление, правка, удаление и, возможно, какие-то другие), имеет имя, совпадающее с названием этой операции, вычисляет некоторое условие и, в зависимости от результата вычисления, разрешает или запрещает выполнение операции. Эти методы полностью аналогичны функциям-шлюзам — они принимают те же параметры и возвращают такой же результат логического типа.

Файлы с классами политик располагаются в папке app\Policies, помещаются в пространство имен `App\Policies`, а их имена, по соглашению, должны заканчиваться словом Policy. Эти классы не имеют родителя.

Средства прототипирования Laravel позволят нам создать класс политики. Для этого нужно открыть консоль OpenServer, перейти в папку проекта и отдать команду вида:

```
php artisan make:policy <имя класса политики>
[--model=<имя класса модели>]
```

Если указан параметр `--model`, будет создан класс политики, предназначенный для разграничения доступа к данным модели, чье *имя класса* было задано. Созданный класс политики уже будет содержать набор методов `view`, `create`, `update` и `delete`, соответствующих операциям просмотра, создания, правки и удаления записи модели. Если этот параметр опущен, будет создан «пустой» класс политики, без всяких методов.

Листинг 33.2 показывает полный код только что сгенерированного класса политики `ArticlePolicy`, задающий условия доступа к данным модели `Article`, т. е. к списку статей.

Листинг 33.2

```
<?php
namespace App\Policies;

use App\User;
use App\Article;
use Illuminate\Auth\Access\HandlesAuthorization;
```

```
class ArticlePolicy
{
  use HandlesAuthorization;

  public function view(User $user, Article $article)
  {
    //
  }

  public function create(User $user)
  {
    //
  }

  public function update(User $user, Article $article)
  {
    //
  }

  public function delete(User $user, Article $article)
  {
    //
  }
}
```

Как уже говорилось, метода классы политики принимают те же параметры, что и функции-шлюзы, и возвращают результат аналогичного типа. Вот пример метода политики `update`, представляющего операцию `"update"` и позволяющего править статьи только их автору:

```
public function update(User $user, Article $article) {
  return $article->author == $user->id;
}
```

Есть возможность переопределить поведение политики на уровне всего класса, объявив в нем публичный метод `before`. Этот метод будет вызываться перед выполнением любого другого метода класса политики. Он должен принимать два параметра: объект класса модели, представляющий текущего пользователя, и строку с обозначением выполняемой операции: `"view"`, `"create"`, `"update"`, `"delete"` и др. В качестве результата он может вернуть:

- `true` — тогда операция будет разрешена, независимо от результата выполнения соответствующего ей метода;
- `false` — тогда операция будет запрещена, независимо от результата выполнения соответствующего ей метода;
- `null` — обычное выполнение, т. е. будет вызван метод, соответствующий операции, и возвращенный им результат разрешит или запретит эту операцию.

Вот пример кода метода `before`, разрешающий выполнение всех операций пользователю, имеющему роль "администратор":

```
public function before($user, $operation) {
  if ($user->role == "admin") {
    return true;
  } else {
    return null;
  }
}
```

При этом, если пользователь имеет роль, отличную от администраторской, его права на выполнение тех или иных операций будут определять соответствующие этим операторам методы класса политики.

После создания класса политики следует связать его с моделью и зарегистрировать во фреймворке. Для этого нужно объявить в классе провайдера `AuthServiceProvider` (о нем рассказывалось ранее) защищенное свойство `policies`, значением которого должен быть ассоциативный массив. Ключами элементов этого массива будут полные имена классов моделей, а значениями элементов — полные имена классов связываемых с моделями политик.

Листинг 33.3 показывает фрагмент кода класса `AuthServiceProvider`, в котором указана связь между моделью `Article` и политикой `ArticlePolicy`.

Листинг 33.3

```
class AuthServiceProvider extends ServiceProvider {
  protected $policies = ['App\Article' => 'App\Policies\ArticlePolicy'];
  . . .
}
```

Произвести проверку, может ли пользователь выполнить какую-либо операцию над той или иной записью, можно несколькими способами:

- в выражении, задающем маршрут, — указав в вызове метода `middleware` (см. *главу 27*) посредник `can` в формате:

  `can:`*`<операция>`*`,`*`<параметр>`*

  Операция должна совпадать с именем метода, объявленного в классе политики. Значение *параметра* берется из интернет-адреса. Им должен быть объект класса модели, представляющий запись, — ведь по классу модели Laravel определяет политику, к которой следует обратиться. Следовательно, для маршрута необходимо задействовать механизм внедрения модели (см. *главу 29*).

  Если пользователь не имеет права выполнить указанную операцию, Laravel выведет страницу с сообщением об ошибке.

  В следующем примере мы проверяем, может ли текущий пользователь выполнять операцию "`update`" над статьей, хранящейся в параметре `article`:

```
Route::get("/articles/{article}/edit", "ArticleController@input")
->middleware("can:update,article");
```

Как уже говорилось ранее, Laravel определяет модель и, соответственно, политику, к которой нужно обратиться, проверяя тип значения параметра. Если действие не принимает параметра (как, например, действие, создающее новую запись), в качестве параметра следует указать полное имя класса модели. Пример:

```
Route::get("/articles/create", "ArticleController@input")
->middleware("can:create,App\Article");
```

- в коде конструктора контроллера — указав в вызове метода `middleware` тот же посредник `can`:

```
public function __construct() {
  $this->middleware("can:manipulate,App\User");
}
```

- в коде действия контроллера, вызвав один из следующих методов класса модели `User`, представляющей зарегистрированного пользователя:
  - *<пользователь>*`->can(`*<операция>*`, `*<аргументы, передаваемые методу политики и разделенные запятыми>*`)` — возвращает `true`, если пользователь может выполнить указанную операцию, и `false` в противном случае. Пример:

```
if (auth()->user()->can("update", $article)) {
  // Текущий пользователь может исправить эту статью
}
```

  - *<пользователь>*`->cant(`*<операция>*`, `*<аргументы, передаваемые методу политики и разделенные запятыми>*`)` — возвращает `true`, если пользователь не может выполнить указанную операцию, и `false` в противном случае;
  - *<пользователь>*`->cannot(`*<операция>*`, `*<аргументы, передаваемые методу политики и разделенные запятыми>*`)` — то же самое, что и `cant`.
- также в коде действия контроллера, вызвав поддерживаемый классом контроллера метод `authorize`:

```
<контроллер>->authorize(<операция>,
<аргументы, передаваемые методу политики и разделенные запятыми>)
```

Если пользователь не имеет права выполнить указанную *операцию*, этот метод прервет выполнение кода метода-действия и выведет страницу с сообщением об ошибке.

В следующем примере мы проверяем, может ли текущий пользователь создавать новые статьи. Обратите внимание, что в качестве аргумента, передаваемого методу политики, мы указали полное имя класса модели, извлеченное из статического свойства `class`:

```
public function save() {
  . . .
  $this->authorize("create", Article::class);
}
```

Ограничение доступа на основе запросов форм

В *главе 32* мы познакомились с запросами форм, выполняющими автоматическую валидацию занесенных в Web-форму данных. Они также могут проверять, имеет ли текущий пользователь право на выполнение какой-либо операции с записями модели, — и также автоматически. Если пользователь не имеет права выполнять операцию, будет выведена страница с сообщением об ошибке.

Классы запросов форм поддерживают, помимо метода `rules`, еще и метод `authorize` (с которым мы уже немного знакомы). Он не принимает параметров, а в качестве результата должен вернуть `true`, если пользователь имеет право выполнить операцию, и `false`, если у него нет такого права.

Листинг 33.4 показывает фрагмент кода класса `CategoryRequest`, который представляет собой запрос формы и в котором проверяется, может ли текущий пользователь работать с категориями. Здесь создавать, править и удалять категории разрешено лишь пользователям, имеющим роли редактора или администратора.

Листинг 33.4

```
class CategoryRequest extends FormRequest {
  public function authorize() {
    $user = auth()->user();
    return ($user->role == "e") || ($user->role == "m");
  }
    . . .
}
```

Ограничение доступа в шаблонах

Если пользователь не имеет права выполнять какие-либо операции и уж, тем более, если он не выполнил вход на сайт (т. е. является гостем), имеет смысл скрыть элементы управления (гиперссылки или кнопки), с помощью которых выполняются эти операции. Например, если пользователь не имеет права на правку и удаление статей, не следует выводить на экран гиперссылки, ведущие на соответствующие действия.

Специально для такого случая программный модуль Blade, выполняющий обработку шаблонов и входящий в состав Laravel, поддерживает две дополнительные команды встроенного языка:

- ```
  @can(<операция 1>, <аргументы 1>)
    <блок can 1>
  [@elsecan(<операция 2>, <аргументы 2>)
    <блок can 2>
  . . .
  @elsecan(<операция n>, <аргументы n>)
    <блок can n>]
  @endcan
  ```
  ```

Если текущий пользователь имеет право выполнить `операцию 1`, будет выполнен `блок can 1`. Если текущий пользователь имеет право выполнить `операцию 2`, будет выполнен `блок can 2`, и т. д.

- ```
  @cannot(<операция 1>, <аргументы 1>)
    <блок cannot 1>
  [@elsecannot(<операция 2>, <аргументы 2>)
    <блок cannot 2>
  . . .
  @elsecannot(<операция n>, <аргументы n>)
    <блок cannot n>]
  @endcannot
  ```

  Если текущий пользователь не имеет права выполнить `операцию 1`, будет выполнен `блок cannot 1`. Если текущий пользователь не имеет права выполнить `операцию 2`, будет выполнен `блок cannot 2`, и т. д.

  В обоих случаях и `операции`, и `аргументы` задаются в виде строк.

В следующем примере мы выводим гиперссылку, указывающую на страницу создания категории, только если пользователь имеет право создавать категории:

```
@can ("create", "App\Category")
  <a href="{{ action("CategoryController@create") }}">Создать
  категорию</a>
@endcan
```

Вывод сведений о текущем пользователе

Часто приходится выводить на экран или обрабатывать в программном коде сведения о текущем пользователе: его имя, адрес электронной почты, роль и др. А еще чаще приходится проверять, выполнил ли посетитель вход на сайт.

Методы, которые будут описаны далее, могут быть вызваны у фасада `Auth` или у результата, возвращенного не принимающей аргументов функцией `auth`.

Не принимающий аргументов метод `user` возвращает объект класса модели `User`, представляющий текущего пользователя. Пример:

```
use Illuminate\Support\Facades\Auth;
. . .
$user = Auth::user();
```

Мы можем получить сведения о пользователе, применив те же самые приемы, что использовались для доступа к полям модели. Пример:

```
$userName = $user->name;
$email = $user->email;
```

Не принимающий аргументов метод `check` возвращает `true`, если текущий пользователь выполнил вход на сайт, и `false`, если он является гостем.
```

В следующем примере, если пользователь выполнил вход на сайт, мы выводим абзац с его именем:

```
@if (auth()->check())
  <p>Пользователь: {{ auth()->user()->name }}</p>
@endif
```

А не принимающий аргументов метод `guest`, наоборот, возвращает `true`, если текущий пользователь является гостем, и `false`, если он выполнил вход на сайт.

Не принимающий аргументов метод `id` возвращает номер текущего пользователя или `null`, если текущий пользователь — гость.

Настройка встроенной подсистемы разграничения доступа

Разумеется, встроенная в Laravel подсистема разграничения доступа предусматривает возможность настройки. Более того, нам, скорее всего, придется заняться ее настройками.

Базовые настройки

Сначала рассмотрим базовые настройки, для осуществления которых нам будет достаточно полученных ранее знаний.

- Прежде всего, мы можем изменить шаблоны всех четырех страниц, равно как и шаблон, от которого они наследуются. Единственное, что нужно иметь в виду: не следует изменять имена элементов управления, в которые заносятся данные (адрес электронной почты, пароль и проч.).
- Предлагаемая Laravel реализация процедуры выхода с сайта имеет ключевой недостаток. Если мы посмотрим на код маршрута, указывающего на действие, которое выполняет выход:

  ```
  Route::post('logout', 'Auth\LoginController@logout')->name('logout');
  ```

  то увидим, что этот маршрут предполагает отправку данных методом POST. Это значит, что нам придется создать отдельную страницу выхода с сайта с формой, в которой указать заданный в этом маршруте интернет-адрес и поместить кнопку отправки данных. Это не всегда удобно.

  Однако мы можем добавить в код маршрутизатора еще и такой маршрут:

  ```
  Route::get('logout', 'Auth\LoginController@logout');
  ```

  После чего для выполнения процедуры выхода будет достаточно создать на какой-либо странице гиперссылку, ведущую на интернет-адрес **/logout**. Процедура выхода будет выполнена после перехода по этой гиперссылке. Так гораздо удобнее, поскольку нам не придется создавать еще одну страницу для выхода с сайта.

- По умолчанию после регистрации, входа и сброса пароля будет выполнено перенаправление на маршрут с именем `home`. Но мы можем указать другой маршрут, записав его в качестве значения защищенного свойства `redirectTo` классов контроллеров `RegisterController`, `LoginController` и `ResetPasswordController`:

  ```
  class LoginController extends Controller {
    . . .
    protected $redirectTo = '/';
    . . .
  }
  ```

  Если интернет-адрес, на который выполняется перенаправление, должен генерироваться программно, мы можем вместо свойства `redirectTo` объявить в перечисленных ранее классах контроллеров защищенный, не принимающий аргументов метод `redirectTo`.

  В следующем примере мы выполняем перенаправление на страницу списка подкатегорий:

  ```
  class LoginController extends Controller {
    . . .
    protected function redirectTo() {
      return redirect()->action("SubcategoryController@index");
    }
  }
  ```

- После выхода с сайта перенаправление по умолчанию выполняется на интернет-адрес /, т. е. на главную страницу. Если нам нужно, чтобы после выхода с сайта выполнялось перенаправление по какому-либо другому интернет-адресу, мы объявим в классе `LoginController` публичный, не принимающий аргументов метод `logout`, который будет выполнять выход с сайта с последующим перенаправлением.

  В следующем примере для выполнения выхода с сайта применяется не принимающий аргументов и не возвращающий результата метод `logout` фасада `Auth`:

  ```
  use Illuminate\Support\Facades\Auth;
  . . .
  class LoginController extends Controller {
    . . .
    public function logout() {
      Auth::logout();
      return redirect("/subcategories");
    }
  }
  ```

> ***Внимание!***
>
> К сожалению, при попытке открытия страницы, доступ к которой ограничен, всегда выполняется перенаправление на интернет-адрес **/login**. Автор книги не нашел способа указать другой интернет-адрес для такого случая.

- По умолчанию встроенная подсистема разграничения доступа использует для идентификации пользователя поле `email` таблицы `users`, хранящее адрес электронной почты. Мы можем использовать для этого другое поле, например, `name`, которое также присутствует в таблице `users` и хранит имя пользователя. Для этого следует добавить в код класса контроллера `LoginController` публичный метод `username`, которые не принимает аргументов и возвращает строку с именем нужного поля:

```
class LoginController extends Controller {
  . . .
  public function username() {
    return "name";
  }
  . . .
}
```

- Сообщения об ошибках на странице регистрации нового пользователя по умолчанию выводятся по-английски. Русскоязычные сообщения можно указать в вызове метода `make` фасада `Validator`, который присутствует в коде защищенного метода `validator` класса `RegisterController`. Пример:

```
protected function validator(array $data) {
  return Validator::make($data, [
    . . .
  ], [
    'name.required' => 'Введите имя пользователя',
    'name.max' => 'Имя пользователя должно быть не длиннее ' .
    '255 символов',
    . . .
  ]);
}
```

- Задать свои собственные сообщения об ошибках, которые выводятся на странице входа на сайт, несколько сложнее.

  Прежде всего следует открыть файл AuthenticatesUsers.php с кодом трейта `AuthenticatesUsers`, хранящийся в папке vendor\laravel\framework\src\illuminate\Foundation\Auth, и скопировать в класс контроллера `LoginController` объявления методов `validateLogin` и `sendFailedLoginResponse`.

  - В коде метода `validateLogin` присутствует вызов метода `validate`, в котором можно указать третьим аргументом ассоциативный массив с сообщениями об ошибках ввода.
  - В коде метода `sendFailedLoginResponse` присутствует вызов метода `withErrors`, в аргументе которого (им выступает ассоциативный массив) можно указать текст сообщения, выводящегося на экран, если пользователь с указанным адресом электронной почты отсутствует в списке.

  Подробнее о методах `validate` и `withErrors` рассказывалось в *главе 32*.

Помимо этого, следует добавить в файл с объявлением класса `LoginController` выражение импортирования класса `Illuminate\Http\Request`.

Пример исправленного кода контроллера `LoginController`, в котором заданы собственные сообщения об ошибках, приведен в листинге 33.5.

Листинг 33.5

```
. . .
use Illuminate\Http\Request;

class LoginController extends Controller {
  . . .
  protected function validateLogin(Request $request) {
    $this->validate($request, [
      $this->username() => 'required', 'password' => 'required',
    ], [
      $this->username() . '.required' => 'Введите адрес электронной ' .
      'почты',
      'password.required' => 'Введите пароль',
    ]);
  }

  protected function sendFailedLoginResponse(Request $request) {
    return redirect()->back()
    ->withInput($request->only($this->username(), 'remember'))
    ->withErrors([
      $this->username() => 'Пользователь с таким адресом электронной ' .
      'почты отсутствует в списке',
    ]);
  }
}
```

- Чтобы указать собственные сообщения об ошибках для страницы отправки электронного письма с гиперссылкой, указывающей на страницу сброса пароля, нужно открыть файл vendor\laravel\framework\src\illuminate\Foundation\Auth\SendsPasswordResetEmails.php, где хранится код трейта `SendsPasswordResetEmails`, и скопировать оттуда в класс контроллера `ForgotPasswordController` объявления методов `sendResetLinkEmail`, `sendResetLinkResponse` и `sendResetLinkFailedResponse`.
  - В коде метода `sendResetLinkEmail` есть вызов метода `validate`, где третьим аргументом можно задать ассоциативный массив сообщений об ошибках ввода.
  - В коде метода `sendResetLinkResponse` присутствует вызов метода `with`, и в нем можно указать свой текст сообщения об успешной отправке электронного письма. (Метод `with` описан в *главе 30*.)

- В коде метода `sendResetLinkFailedResponse` есть вызов метода `withErrors`, в котором можно задать текст сообщения об отсутствии заданного адреса электронной почты в списке пользователей.

Помимо этого, следует добавить в файл с объявлением класса `ForgotPasswordController` выражения импортирования классов `Illuminate\Http\Request` и `Illuminate\Support\Facades\Password`.

Пример исправленного кода контроллера `ForgotPasswordController`, в котором заданы собственные сообщения об ошибках, приведен в листинге 33.6.

Листинг 33.6

```
. . .
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

class ForgotPasswordController extends Controller {
  . . .
  public function sendResetLinkEmail(Request $request) {
    $this->validate($request, ['email' => 'required|email'], [
      'email.required' => 'Введите адрес электронной почты',
      'email.email' => 'Введите корректный адрес электронной почты',
    ]);
    . . .
  }

  protected function sendResetLinkResponse($response) {
    return back()->with('status', 'Письмо со ссылкой на страницу ' .
    'сброса пароля успешно отправлено');
  }

  protected function sendResetLinkFailedResponse(Request $request,
  $response) {
    return back()->withErrors(
    ['email' => 'Указанный адрес электронной почты в списке ' .
    'отсутствует']);
  }
}
```

❒ Чтобы указать собственные сообщения об ошибках для страницы собственно сброса пароля, следует открыть файл vendor\laravel\framework\src\illuminate\Foundation\Auth\ResetsPasswords.php, где хранится код трейта `ResetsPasswords`, и скопировать оттуда в класс контроллера `ResetPasswordController` объявления методов `validationErrorMessages`, `sendResetResponse` и `sendResetFailedResponse`.

- Метод `validationErrorMessages` должен возвращать ассоциативный массив сообщений об ошибках ввода, заданный в том же формате, что указывается третьим аргументом в методе `validate` (изначально он возвращает пустой массив).

* В коде метода `sendResetResponse` есть вызов метода `with`, в котором можно указать сообщение об успешном сбросе пароля.
* В коде метода `sendResetFailedResponse` присутствует вызов `withErrors`, в котором можно указать сообщение о том, что заданный адрес электронной почты отсутствует в списке.

Помимо этого, следует добавить в файл с объявлением класса `ResetPasswordController` выражение импортирования класса `Illuminate\Http\Request`.

Пример исправленного кода контроллера `ResetPasswordController` с заданными сообщениями об ошибках показан в листинге 33.7.

Листинг 33.7

```
. . .
use Illuminate\Http\Request;

class ResetPasswordController extends Controller {
  . . .
  protected function validationErrorMessages() {
    return [
      'email.required' => 'Укажите адрес электронной почты',
    ];
  }

  protected function sendResetResponse($response) {
    return redirect($this->redirectPath())
    ->with('status', 'Сброс пароля успешно выполнен');
  }

  protected function sendResetFailedResponse(Request $request,
  $response) {
    return redirect()->back()
    ->withInput($request->only('email'))
    ->withErrors(['email' => 'Указанный адрес электронной почты в ' .
    'списке отсутствует']);
  }
}
```

Модификация списка пользователей

Миграция, формирующая таблицу `users`, хранящую список зарегистрированных пользователей, создает в этой таблице следующие поля:

- `id` — поле счетчика, хранящее номер пользователя;
- `name` — имя пользователя;

- `email` — адрес электронной почты пользователя (уникальное);
- `password` — пароль пользователя (хранится в зашифрованном виде);
- `remember_token` — ключ для восстановления пароля;
- `created_at` и `updated_at` — хранят дату и время, соответственно, создания и последнего изменения записи.

Если нам понадобится сохранить в таблице, помимо этого, еще и другие данные, например роль пользователя, мы вольны добавить в код миграции выражения, создающие дополнительные поля. Только это следует делать перед выполнением миграции.

Мы также можем реализовать вывод списка пользователей, хранящегося в таблице `users`, для какой-либо цели.

Миграция, создающая таблицу `password_resets`, использующуюся в процедуре выполнения пароля, создает следующие поля:

- `email` — адрес электронной почты пользователя;
- `token` — ключ для восстановления пароля;
- `created_at` — хранит дату и время создания записи (может хранить `null`).

Настройка писем, отправляемых действием восстановления пароля

При выполнении процедуры восстановления пароля соответствующее действие контроллера `ForgotPasswordController` отправляет по занесенному посетителем адресу электронное письмо. В тексте этого письма присутствует гиперссылка, перейдя по которой, посетитель выполнит сброс своего пароля.

Изначально это письмо написано по-английски. Однако мы можем задать для него свой собственный текст. Но для этого придется повозиться...

Создание оповещения

Механизм отправки подобных писем в Laravel реализован через так называемые *оповещения*, которые можно рассматривать как короткие письма, сообщающие о чем-либо, и которые могут отправляться как по электронной почте, так и другими путями, например, в SMS. Чтобы указать свой собственный текст письма, нам придется создать новое оповещение.

Оповещения Laravel реализованы в виде классов, находящихся в пространстве имен `App\Notifications` и являющихся потомками класса `Notification`. Файлы с кодом классов оповещений хранятся в папке app\Notifications.

Для создания оповещения мы можем воспользоваться средствами прототипирования Laravel. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду формата:

```
php artisan make:notification <имя класса оповещения>
```

Листинг 33.8 показывает полный код только что созданного класса оповещения `PasswordReset`.

Листинг 33.8

```
<?php
namespace App\Notifications;
use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class PasswordReset extends Notification
{
  use Queueable;

  public function __construct()
  {
    //
  }

  public function via($notifiable)
  {
    return ['mail'];
  }

  public function toMail($notifiable)
  {
    return (new MailMessage)
                ->line('The introduction to the notification.')
                ->action('Notification Action',
                'https://laravel.com')
                ->line('Thank you for using our application!');
  }

  public function toArray($notifiable)
  {
    return [
      //
    ];
  }
}
```

Первое, что нам нужно сделать, — объявить в этом классе публичное свойство, предназначенное для хранения ключевого значения, участвующего в процедуре сброса пароля. Пример:

```
class PasswordReset extends Notification {
  public $token;
  . . .
}
```

Далее в классе нужно изменить код конструктора таким образом, чтобы он принимал единственный параметр — это самое ключевое значение — и заносил его в только что добавленное свойство. Пример:

```
public function __construct($token) {
  $this->token = $token;
}
```

После этого мы укажем текст отправляемого оповещения. Для этого мы модифицируем код метода `toMail`, который, собственно, и формирует текст оповещения, предназначенного для отправки по электронной почте. Как видно из листинга 33.5, этот метод создает объект класса `MailMessage`, представляющий электронное письмо, которое и станет оповещением, задает его содержание и возвращает в качестве результата.

Для указания содержания письма можно использовать методы класса `MailMessage`, перечисленные далее. Все эти методы возвращают в качестве результата тот же самый объект, у которого были вызваны, благодаря чему мы можем вызывать следующий метод у результата, возвращенного предыдущим.

- `<письмо>->subject(<строка>)` — задает *строку* в качестве темы письма;
- `<письмо>->greeting(<строка>)` — задает *строку* в качестве заголовка письма;
- `<письмо>->line(<строка>)` — добавляет *строку* к содержанию письма;
- `<письмо>->with(<строка>|<массив строк>)` — то же, что и `line`, только позволяет добавить к содержанию сразу несколько строк, взятых из указанного *массива*;
- `<письмо>->action(<строка>, <интернет-адрес>)` — создает из *строки* гиперссылку, указывающую на *интернет-адрес*;
- `<письмо>->from(<адрес отправителя>[, <имя отправителя>])` — указывает у письма *адрес* и *имя* отправителя.

В листинге 33.9 показан код метода `toMail`, который формирует письмо-оповещение о сбросе пароля.

Листинг 33.9

```
public function toMail($notifiable) {
  return (new MailMessage)
  ->greeting("Уважаемый пользователь!")
  ->line("Вы получили это письмо, поскольку произвели процедуру " .
  "сброса пароля.")
  ->action("Сбросить пароль", url("password/reset", $this->token))
  ->line("Если вы не выполняли сброс пароля, ничего не " .
  "предпринимайте.")
  ->from("admin@supersite.ru", "Администрация");
}
```

Осталось рассмотреть метод `via`, который указывает для оповещения способы отправки. Он должен вернуть массив, элементами которого станут строковые обозначения доступных в Laravel способов. В изначальной реализации он возвращает массив со строкой `"mail"`, указывая для оповещения единственный способ отправки — по электронной почте. Так что модифицировать код этого метода нам не нужно.

Подготовка шаблона для оповещения

Для формирования оповещения о сбросе пароля применяется особый шаблон, поставляемый в составе фреймворка. Но мы можем указать для такого рода оповещений другой шаблон.

Проще всего сделать копию стандартного шаблона, воспользовавшись все теми же средствами прототипирования, после чего модифицировать его. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду:

```
php artisan vendor:publish --tag=laravel-notifications
```

В папке resources\views\vendor\notifications будут созданы два шаблона:

- email-plain.blade.php — шаблон для обычного текстового письма;
- email.blade.php — шаблон для письма в формате HTML.

Собственно, в этих шаблонах нам нужно лишь найти фрагменты текста, написанные по-английски, и заменить их на русскоязычные. Например, вот такой фрагмент:

```
echo 'Regards,', "\n";
echo config('app.name'), "\n";
```

заменить на:

```
echo 'С наилучшими пожеланиями,', "\n";
echo "администрация сайта.", "\n";
```

В *главе 39* правка шаблонов электронных писем такого рода будет рассмотрена более подробно.

Отправка оповещений

Теперь нам нужно написать код, который выполнит отправку оповещений, созданных на основе написанного нами ранее класса. Для этого следует добавить в класс модели `User` публичный метод `sendPasswordResetNotification`. Он должен принимать в качестве параметра ключевое значение, применяемое для сброса пароля, и не возвращать результата.

В коде этого метода мы можем для отправки оповещения использовать метод `notify` класса `User`:

```
<модель>->notify(<объект класса оповещения>)
```

Пример:

```
public function sendPasswordResetNotification($token) {
  $this->notify(new PasswordReset($token));
}
```

По умолчанию адрес электронной почты будет извлекаться из поля `email` модели `User`. Если нам нужно указать другой адрес, мы добавим в класс модели публичный метод `routeNotificationForMail`. Он не должен принимать параметров и должен возвращать в качестве результата строку с адресом электронной почты.

В следующем примере мы указываем в качестве адреса электронной почты значение поля `user_email` модели:

```
public function routeNotificationForMail() {
  return $this->user_email;
}
```

Использование CAPTCHA. Библиотека Captcha for Laravel

Если мы собираемся дать возможность любому посетителю, в том числе и гостю, заносить в базу данных сайта какую-либо информацию (например, комментарии к статьям), нам придется приготовиться к настоящему нашествию программ автоматической рассылки спама. Такие программы сами отыскивают формы для ввода комментариев и заполняют сайт огромным количеством мусорных сообщений.

К счастью, есть замечательный способ раз и навсегда избавиться от этой напасти — использовать *CAPTCHA* (Completely Automated Public Turing test to tell Computers and Humans Apart, полностью автоматизированный публичный тест Тьюринга для различения компьютеров и людей). Он представляет собой графическое изображение с искаженным и «зашумленным» текстом[1], обычно бессмысленным набором букв и цифр, который нужно занести в расположенное рядом поле ввода. Если оба текста совпадут, значит комментарий оставил человек, а не программа, и его можно сохранить в базе.

В составе Laravel не имеется никаких программных компонентов для поддержки CAPTCHA. Однако мы можем загрузить и установить дополнительную библиотеку *Captcha for Laravel*, реализующую его поддержку.

Установка

Чтобы установить библиотеку Captcha for Laravel, достаточно открыть консоль OpenServer, перейти в папку проекта и отдать команду:

```
composer require mews/captcha
```

[1] «Зашумленный» текст содержит так называемый «мусор»: штрихи, точки, кляксы, а также отображает символы, расположенные близко друг от друга, «слипшиеся», наклоненные, развернутые. Человек сможет такой текст прочитать, а компьютерная программа — нет.

Через некоторое время эта библиотека (а также все библиотеки, которые она использует в работе) будет загружена и установлена.

> **ВНИМАНИЕ!**
>
> Библиотека Captcha for Laravel требует, чтобы в составе PHP присутствовал модуль расширения *php_gd2.dll*. Впрочем, в PHP, поставляемой в пакете OpenServer, этот модуль есть.

Чтобы только что установленная Captcha for Laravel успешно заработала, нам потребуется занести изменения в настройки проекта. Эти настройки задаются в файле config\app.php.

Сначала найдем в нем параметр `providers`. Это значение представляет собой большой массив из полных имен классов провайдеров, из которых состоит фреймворк. Нам нужно добавить в этот массив новый пункт — полное имя класса `CaptchaServiceProvider`, который находится в пространстве имен `Mews\Captcha`. Вот так:

```
'providers' => [
  . . .
  Mews\Captcha\CaptchaServiceProvider::class,
]
```

Далее найдем там же параметр `aliases`, указывающий короткие имена для классов-фасадов. Его значение — ассоциативный массив, ключами элементов которого являются короткие имена фасадов, а значениями элементов — полные имена этих же классов. Добавим в этот массив следующий элемент:

```
'aliases' => [
  . . .
  'Captcha' => 'Mews\Captcha\Facades\Captcha',
]
```

На этом установка библиотеки закончена.

Настройка

Прежде чем указывать настройки библиотеки Captcha for Laravel, вынесем ее конфигурационный файл в папку config. Для этого откроем консоль OpenServer, перейдем в папку проекта, наберем и выполним команду:

```
php artisan vendor:publish
```

В результате в папке config появится файл captcha.php, хранящий параметры библиотеки.

Код, находящийся в этом файле, представляет собой объявление ассоциативного массива. Ключи его элементов задают название одной из конфигураций CAPTCHA, изначально поддерживаемых библиотекой: `"default"` (используется по умолчанию), `"flat"` («плоская»), `"mini"` (уменьшенных размеров) и `"inverse"` («инверсная»). Значения элементов — суть наборы параметров, описывающих каждую из

этих конфигураций. Эти значения также представляют собой ассоциативные массивы, где ключи элементов — названия параметров, а значения элементов — значения параметров.

Поддерживаемые библиотекой Captcha for Laravel параметры приведены в табл. 33.1.

Таблица 33.1. *Параметры CAPTCHA, поддерживаемые Captcha for Laravel*

Параметр	Описание	Значение по умолчанию
`length`	Длина текста, выводимого в CAPTCHA, в символах	5
`sensitive`	Если `true`, проверка текста CAPTCHA будет выполняться с учетом регистра символов, если `false` — без учета	`false`
`width`	Ширина CAPTCHA в пикселах	120
`height`	Высота CAPTCHA в пикселах	36
`quality`	Качество графического изображения в %	90
`lines`	Количество линий, выводимых для создания «шума»	3
`bgImage`	Если `true`, выводится фоновое изображение, если `false` — не выводится	`true`
`fontColors`	Массив цветов символов. Если не указан, символы будут выведены черным цветом	`[]`
`bgColor`	Цвет фона, заданный в стандарте CSS	`#ffffff`
`invert`	Если `true`, будет сформировано инверсное изображение, если `false` — обычное	`false`
`fonts`	Массив шрифтов, которыми будут выведены отдельные символы. Доступные для указания шрифты: ABeeZee, Asab, Khand, Open Sans, Roboto и Ubuntu. Если не указан, символы будут выведены шрифтом по умолчанию, выбранным самой библиотекой	`[]`
`angle`	Угол поворота символов в градусах	15
`contrast`	Уровень контрастности. Положительные значения указывают увеличение контрастности, отрицательные — уменьшение	0
`sharpen`	Степень резкости изображения	0
`blur`	Степень размытости изображения	0

В следующем примере мы указываем для конфигурации CAPTCHA по умолчанию текст длиной в 8 символов, размеры изображения 400×200 пикселов и учет регистра символов при проверке:

```
'default'   => [
  'length'      => 8,
  'width'       => 400,
```

```
  'height'      => 200,
  'sensitive'   => true,
],
```

Использование

Сначала необходимо вывести на странице саму CAPTCHA и поле ввода, куда будет заноситься показанный на ней текст. Для вывода CAPTCHA можно использовать одну из следующих функций:

- `captcha_img([<конфигурация>])` — выводит CAPTCHA (тег `<img>` с ее изображением) с применением указанной *конфигурации*. Если таковая не задана, будет использована конфигурация по умолчанию (`"default"`).

  В следующем примере мы выводим изображение CAPTCHA в абзаце с применением конфигурации по умолчанию:

  ```
  <p>{!! captcha_img() !!}</p>
  ```

- `captcha_src([<конфигурация>])` — выводит интернет-адрес, указывающий на изображение CAPTCHA, с применением указанной *конфигурации*. Если таковая не задана, будет использована конфигурация по умолчанию (`"default"`).

  В следующем примере мы выводим тег `<img>`, в качестве значения атрибута `src` которого задаем интернет-адрес изображения CAPTCHA. Последнее будет выведено с применением конфигурации `"flat"`:

  ```
  <p><img src="{{ captcha_src("flat") }}"></p>
  ```

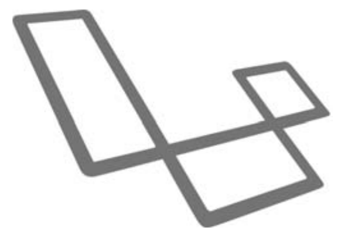
Чтобы выполнить проверку на правильность ввода CAPTCHA в коде контроллера, достаточно при выполнении валидации (подробности — в *главе 32*) для соответствующего POST-параметра указать правило валидации `captcha` (рекомендуется также указать правило `required`). Листинг 33.10 показывает пример кода, который реализует такую проверку.

Листинг 33.10

```
// Шаблон
<p>{!! captcha_img() !!}<br>
Введите символы с изображения: <input name="captcha_code"></p>

// Контроллер
public function save(Request $request) {
  $this->validate($request, ["captcha_code" => "required|captcha"]);
  // Код, выполняющий сохранение записи
}
```

Валидация может быть выполнена с применением другого подхода, например, в запросе формы (которые также описывались в *главе 32*).



ГЛАВА 34

Кэширование

Повышение производительности сайтов, особенно тяжелонагруженных, — в настоящее время весьма сложная задача. Конечно, Web-обозреватель сохраняет (кэширует) все загруженные страницы на дисках клиентского компьютера (*кэширование на стороне клиента*), причем выполняет это безо всякой указки со стороны серверных программ. Но дисковое пространство компьютера клиента ограничено, и рано или поздно ему придется загружать страницы снова. Страницы, которые, вполне возможно, со времени предыдущей загрузки не изменились...

Поэтому Laravel включает в свой состав мощную подсистему *кэширования на стороне сервера*, которая позволяет сохранить произвольные данные на дисках серверного компьютера и использовать их в дальнейшем. Если эти данные получаются в результате сложных вычислений или извлекаются из базы посредством сложных запросов, это может заметно увеличить производительность работы сайта.

Здесь нужно отметить один момент. В отличие от прочих фреймворков (например, Yii), Laravel ничего не кэширует самостоятельно. Нам самим придется написать код, который сохраняет данные в кэше и извлекает их оттуда.

Каждое значение, заносимое в кэш, сохраняется под определенным именем. По этому имени мы в дальнейшем сможем извлечь его.

Настройки кэширования

Laravel позволяет использовать для хранения кэшированных данных несколько вариантов хранилищ: отдельную папку (в этом случае каждое кэшированное значение хранится в отдельном файле), таблицу базы данных (тогда каждое значение будет сохраняться в отдельной записи этой таблицы), оперативную память компьютера и специализированные программы Memcached и Redis. Мы рассмотрим кэширование данных в папке, в таблице базы данных и в оперативной памяти.

Кэширование в папке

Чтобы задать кэширование данных в папке, нужно открыть файл `.env`, где хранятся базовые настройки проекта, найти в этом файле параметр `CACHE_DRIVER` и задать для него значение `file`.

Пример:

```
CACHE_DRIVER=file
```

Для более тонкой настройки следует исправить содержимое настроечного файла cache.php из папки config. В содержимом этого файла нужно найти группу параметров `stores`, а в ней — группу параметров `file`. Вот код этих групп параметров:

```
'stores' => [
    . . .
    'file' => [
        'driver' => 'file',
        'path' => storage_path('framework/cache'),
    ],
    . . .
],
```

Здесь мы можем задать только параметр `path`, указывающий полный путь к папке, в которой, собственно, и будут храниться файлы с кэшированными значениями. По умолчанию, как видим, это папка storage\framework\cache.

Кэширование в таблице базы данных

Чтобы указать фреймворку кэшировать данные в таблице базы данных, нужно открыть файл с базовыми настройками .env и задать для параметра `CACHE_DRIVER` значение `database`. Пример:

```
CACHE_DRIVER=database
```

Для выполнения дополнительной настройки следует открыть файл config\cache.php, найти группу параметров `stores`, а в ней — группу параметров `database`. Вот код, который создает эту группу параметров:

```
'stores' => [
    . . .
    'database' => [
        'driver' => 'database',
        'table' => 'cache',
        'connection' => null,
    ],
    . . .
],
```

Здесь нас также интересует один-единственный параметр `table`, указывающий имя таблицы, которая служит хранилищем для кэшированных данных. По умолчанию это таблица `cache`, и она должна уже существовать в базе данных сайта.

Создать миграцию, генерирующую эту таблицу, мы можем посредством инструментов прототипирования Laravel. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду:

```
php artisan cache:table
```

Спустя несколько секунд миграция будет создана, и нам останется лишь выполнить ее.

Кэширование в оперативной памяти

Если указано кэширование данных в оперативной памяти, Laravel создаст ассоциативный массив, в элементах которого и будет сохранять кэшированные значения. Такой способ кэширования обеспечивает наивысшую производительность, однако при отключении серверного компьютера все содержимое кэша будет потеряно.

Чтобы задать кэширование в памяти, достаточно открыть файл базовых настроек .env, найти параметр `CACHE_DRIVER` и задать для него значение `array`. Пример:

```
CACHE_DRIVER=array
```

Занесение данных в кэш

Просто занести значение в кэш, сохранив его на указанное время, можно вызовом метода `put` фасада `Cache`:

```
Cache::put(<имя>, <значение>, <время хранения>)
```

Имя указывается в виде строки, а *время хранения* — в виде числа в минутах. Метод не возвращает результата.

В следующем примере мы сохраняем в кэше список категорий под именем `cats` на один час:

```
use Illuminate\Support\Facades\Cache;
. . .
$cats = Category::all();
Cache::put("cats", $cats, 60);
```

Метод `putMany` фасада `Cache` сохраняет в кэше сразу несколько значений:

```
Cache::putMany(<список значений>, <время хранения>)
```

Список значений указывается в виде ассоциативного массива, в котором ключи элементов станут именами сохраняемых в кэше значений, а значения элементов — самими сохраняемыми значениями. Этот метод также не возвращает результата.

В следующем примере мы сохраняем в кэше списки категорий и подкатегорий под именами `cats` и `subcats` соответственно на один час:

```
$cats = Category::all();
$subcats = Subcategory::all();
Cache::put(["cats" => $cats, "subcats" => $subcats], 60);
```

Метод `forever` фасада `Cache` сохраняет значение в кэше навсегда:

```
Cache::forever(<имя>, <значение>)
```

Пример:

```
Cache::forever("siteName", "Сайт публикаций");
```

Метод `add` фасада `Cache` сохраняет значение в кэше только в том случае, если оно там еще не существует. Он вызывается так же, как метод `put`, и возвращает `true`, если значение было добавлено в кэш, и `false`, если оно там уже существует. Пример:

```
Cache::add("counter", 0);
```

Для сохранения значений в кэше можно использовать функцию `cache`, вызванную в следующем формате:

```
cache(<список значений>, <время хранения>)
```

Список значений указывается в том же виде, что и при вызове метода `putMany`. Пример:

```
$cats = Category::all();
$subcats = Subcategory::all();
cache(["cats" => $cats, "subcats" => $subcats], 30);
```

Изменение данных, хранящихся в кэше

Два метода — `increment` и `decrement` — фасада `Cache` позволят нам увеличить или уменьшить числовое значение, сохраненное в кэше, на указанную величину:

```
Cache::increment|decrement(<имя>[, <величина>])
```

Если *величина* не указана, она принимается равной единице, и, соответственно, значение с заданным *именем* будет инкрементировано или декрементировано.

Оба метода в качестве результата возвращают уже измененное значение или `false`, если таковое в кэше отсутствует. Примеры:

```
$newCounter = Cache::increment("counter");
Cache::decrement("votes", 2);
```

Получение данных из кэша

Мы можем просто получить сохраненное в кэше значение, получить с одновременным сохранением, если значение в кэше отсутствует, или получить его с последующим удалением из кэша.

Простое получение данных

Чтобы просто получить сохраненное в кэше значение с указанным именем, нужно вызвать метод `get` фасада `Cache`:

```
Cache::get(<имя>[, <значение по умолчанию>])
```

В качестве результата метод возвращает само кэшированное значение. Если таковое отсутствует, возвращается *значение по умолчанию* или `null`, если оно не указано.

В качестве второго аргумента можно указать анонимную функцию, не принимающую параметров и возвращающую значение, которое и будет возвращено методом `get`.

Вот пара примеров:

- извлекаем список категорий, ранее сохраненный в кэше под именем `cats`.

  ```
  $cats = Cache::get("cats");
  ```

- извлекаем список категорий или формируем его заново, если он отсутствует в кэше:

  ```
  $cats = Cache::get("cats", function () {
    return Category:all();
  });
  ```

Метод `many` фасада `Cache` позволяет получить из кэша сразу несколько значений:

```
Cache::many(<список имен>)
```

Список имен указывается в виде массива. Метод возвращает ассоциативный массив, где ключи элементов сформированы на основе имен сохраненных в кэше значений, а значения элементов являются самими этими значениями. Пример:

```
$arr = Cache::many(["cats", "subcats"]);
$cats = $arr["cats"];
$subcats = $arr["subcats"];
```

Метод `has` фасада `Cache` позволяет проверить, существует ли в кэше значение с заданным именем:

```
Cache::has(<имя>)
```

В качестве результата возвращается `true`, если значение с заданным *именем* существует, и `false` в противном случае. Пример:

```
if (Cache::has("subcats")) {
  // Список подкатегорий в кэше существует
}
```

Для извлечения данных из кэша еще можно использовать уже знакомую нам функцию `cache`, вызов которой в этом случае записывается в следующем формате:

```
cache(<имя>[, <значение по умолчанию>])
```

Пример:

```
$cats = cache("cats");
```

Получение данных из кэша с одновременной их записью

Часто возникает необходимость попытаться получить данные из кэша и, если они там отсутствуют, выполнить их занесение туда. Laravel позволяет нам это сделать, написав всего одно выражение.

Метод `remember` фасада `Cache` извлекает из кэша значение с указанным именем или, если таковое не существует, записывает его туда:

```
Cache::remember(<имя>, <время хранения>, <функция, вычисляющая значение>)
```

Если значение с указанным *именем* есть в кэше, метод возвращает его. Если такого значения нет, выполняется *функция, вычисляющая значение*, и возвращенный ей результат будет записан в кэш и возвращен методом `remember` в качестве результата. Эта функция не должна принимать параметров.

В следующем примере мы пытаемся получить из кэша список категорий, сохраненный под именем `cats`. Если таковой в кэше отсутствует, получаем список категорий, отсортированных по именам, и сохраняем его в кэше на 10 минут под тем же именем:

```
$cats = Cache::remember("cats", 10, function () {
  return Category::orderBy("name")->get();
});
```

Метод `sear` фасада `Cache` выполняет ту же задачу, но сохраняет значение в кэше навсегда:

```
Cache::sear(<имя>, <функция, вычисляющая значение>)
```

Пример:

```
$siteName = Cache::sear("siteName", function () {
  return "Сайт публикаций"
});
```

Получение данных из кэша с последующим их удалением

Чтобы получить из кэша значение, а потом удалить его, нужно вызвать метод `pull` фасада `Cache`:

```
Cache::pull(<имя>[, <значение по умолчанию>])
```

Как видим, он вызывается так же, как и метод `get`, и возвращает аналогичный результат. Пример:

```
$subcats = Cache::pull("subcats", []);
```

Удаление данных из кэша

Для удаления значения с заданным именем применяется метод `forget` фасада `Cache`:

```
Cache::forget(<имя>)
```

Метод возвращает `true`, если значение было успешно удалено, и `false` в противном случае. Пример:

```
Cache::forget("cats");
```

Не принимающий аргументов и не возвращающий результата метод `flush` фасада `Cache` удаляет из кэша все сохраненные там данные. Пример:

```
Cache::flush();
```

Работа с другим хранилищем

Если описанные ранее методы вызваны непосредственно у фасада `Cache`, они выполняют манипуляции с хранилищем, выбранным по умолчанию. Это хранилище указывается в параметре `CACHE_DRIVER` файла базовых настроек .env.

Однако мы можем обратиться к другому хранилищу. Для этого нужно предварительно вызвать у фасада `Cache` метод `store`:

```
Cache::store(<обозначение хранилища>)
```

В качестве обозначения хранилища указываются строки `"file"` (папка), `"database"` (таблица в базе данных) или `"array"` (оперативная память).

А все методы, описанные ранее и предназначенные для манипулирования сохраненными в кэше данными, следует вызывать у результата, возвращенного методом `store`.

Вот пара примеров:

- сохраняем в кэше список категорий, указав в качестве хранилища оперативную память:

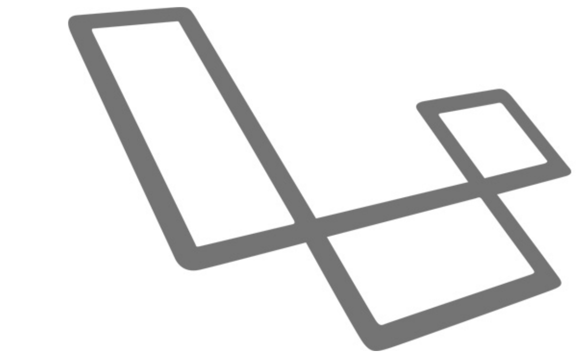
  ```
  $cats = Category::all();
  Cache::store("array")->put("cats", $cats, 30);
  ```

- извлекаем из кэша, в качестве которого используется оперативная память, список категорий:

  ```
  $cats = Cache::store("array")->get("cats");
  ```

ЧАСТЬ III

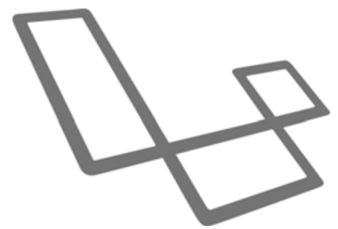
ПРАКТИКА РАЗРАБОТКИ: СОЗДАНИЕ WEB-САЙТА ЭЛЕКТРОННЫХ ПУБЛИКАЦИЙ



III. РАЗДЕЛ 6

Разработка Web-сайта — сводим все воедино

Глава 35.	Планирование и предварительные действия
Глава 36.	Создание дизайна Web-страниц
Глава 37.	Интерактивные элементы
Глава 38.	Статические Web-страницы
Глава 39.	Разграничение доступа и список пользователей
Глава 40.	Категории и подкатегории
Глава 41.	Статьи. Поддержка BBCode
Глава 42.	Комментарии
Глава 43.	Хранилище файлов



ГЛАВА 35

Планирование и предварительные действия

До этого момента мы занимались теорией и выполняли своего рода лабораторные работы, чтобы закрепить полученные знания. Но теперь теория кончилась, и настало время практики.

Мы создадим полностью работоспособный Web-сайт службы электронных публикаций. На нем сторонние авторы смогут публиковать свои статьи, посвященные различным темам. Для рубрикации статей мы предусмотрим набор категорий, каждая из которых будет включать подкатегории. Посетители смогут оставлять комментарии к статьям, причем для этого им не потребуется регистрироваться на сайте. Также мы создадим хранилище файлов, куда авторы статей смогут выгружать файлы с изображениями, аудио- и видеороликами и сторонними данными — например, архивами и дистрибутивами программ, эти файлы будут использованы в качестве иллюстраций и дополнительных материалов.

На примере этого сайта мы рассмотрим более сложные приемы программирования: создание разметки, определение, откуда посетитель зашел на сайт: с обычного компьютера или мобильного устройства, AJAX-подобную выгрузку файлов и др.

Планирование Web-сайта

Но сначала мы займемся планированием нашего будущего сайта. В самом деле, прежде чем что-то делать, необходимо решить, что же мы хотим получить в результате. И решить нужно в самом начале, перед тем, как приниматься за дело. Ведь когда работа сделана, менять что-либо значительно труднее, чем сразу делать все как надо.

Основные этапы планирования Web-сайта

Итак, о чем нужно подумать перед началом работы над сайтом?

1. Прежде всего, четко определить *назначение* сайта. Что он должен делать: рассказывать о ком-либо или о чем-либо, привлекать клиентов, помогать посетителям решать какие-то проблемы, просвещать или развлекать их.

2. Решить, в каком ключе будет выполнен *дизайн* сайта. Будет ли он консервативным, строгим или затейливым. Соответственно, домашняя страничка должна отражать эстетические наклонности автора: развлекательный сайт лучше сделать повеселее, а новостной или сайт с электронными публикациями, подобный нашему, — поскромнее, чтобы пестрота дизайна не заслоняла главное — информацию. На этом этапе лучше всего будет набросать на бумаге, как должна выглядеть та или иная страница.
3. Продумать *логическую структуру сайта* — список входящих в его состав страниц — и, желательно, нарисовать ее. Здесь лучше не изобретать самому велосипед, а посетить какой-нибудь уже существующий и популярный Web-сайт сходной тематики и посмотреть, как он организован. Например, структура сайта электронных публикаций может быть такой:
   - главная страница с краткими сведениями о сайте и его назначении;
   - набор страниц со списками статей, относящихся к различным подкатегориям;
   - набор страниц с собственно статьями и комментариями к ним;
   - служебные страницы (списки категорий, подкатегорий и зарегистрированных пользователей, страницы для добавления, правки и удаления категорий, подкатегорий, статей, комментариев и пользователей, страницы регистрации нового пользователя, входа и выхода);
   - прочие страницы (контакты, политика публикации статей, сведения о разработчиках сайта и др.).
4. Продумать *физическую структуру сайта*, т. е. набор контроллеров и действий, реализующих функциональность сайта, набор папок, где будут храниться выгруженные посетителем файлы, и проч.
5. Продумать систему маршрутизации и формат интернет-адресов, по которым будут располагаться страницы нашего сайта. К ним предъявляется всего одно требование — они должны быть дружественными к пользователю.
6. Проверить, ничего ли мы не забыли. Это последний этап планирования сайта, но не менее важный, чем остальные.

Это — что касается собственно планирования сайта. Теперь поговорим о его дизайне, а затем скажем несколько слов о структуре — логической и физической.

Дизайн Web-сайта

Среди Web-дизайнеров ходит странноватая, на первый взгляд, шутка: дизайн сайта должен быть незаметным. Если перевести ее на более понятный язык, то она станет звучать так: дизайн сайта не должен мешать восприятию информации, которая опубликована на сайте.

Сейчас в моде так называемый «акриловый» дизайн. Это полупрозрачный фон, сквозь который просвечивает какое-либо изображение, и на этом фоне — непрозрачные плашки, возможно, имеющие другой фон, на которых располагаются отдельные элементы разметки («шапка», панель навигации, «поддон» и др.). Плюс

крупные шрифты без засечек, тонкие рамки с острыми или слегка скругленными углами, светлые неконтрастные цветовые схемы, ясное выделение интерактивных элементов (гиперссылок и элементов управления), общая лаконичность и «легкость» дизайна — и получим типичный современный сайт.

Текущая обстановка требует также обеспечения доступности сайтов (об этом уже говорилось в *главе 1*). Это значит, что необходимо предусмотреть, по крайней мере, две редакции одного и того же сайта: для традиционных компьютеров и для мобильных устройств. С учетом того, что в серверных программах мы можем без проблем определить, с какого устройства посетитель зашел на сайт (что и сделаем в *главе 39*), это может быть достигнуто лишь использованием соответствующего родительского шаблона и соответствующих таблиц стилей. (Дополнительно можно использовать медиазапросы, описанные в *главе 15*, — например, для формирования печатных версий страниц.)

Страницы сайта, предназначенные для традиционных компьютеров, будут содержать боковую панель навигации. Она включит в свой состав гиперссылки, ведущие на все основные страницы сайта, — в нашем случае на главную страницу, страницы подкатегорий, на служебные и прочие страницы. Поскольку традиционные компьютеры имеют большие экраны, места для вывода боковой панели навигации будет предостаточно.

На страницах «компьютерной» редакции сайта можно использовать шрифты меньшего размера и более богатое оформление, включающее некоторые графические элементы (скажем, логотип в «шапке»). Также на них можно реализовать подсветку интерактивных элементов при наведении курсора мыши.

Что касается страниц «мобильной» редакции сайта, то шрифты на них следует делать крупнее, оформление — более лаконичным, а подсветка элементов не нужна совсем, т. к. мобильные устройства не имеют мыши. Боковая панель навигации на них лишь будет занимать свободное пространство, поэтому придется предусматривать для посетителей какие-то другие способы навигации. Например, список категорий можно вывести непосредственно на главной странице, а списки подкатегорий — на страницах категорий.

Мы можем разработать дизайн сайта самостоятельно, а можем взять уже готовый, созданный профессиональным Web-дизайнером. Готовые дизайны сайтов различного назначения, выполненные либо в формате Adobe Photoshop, либо в виде уже сверстанных Web-страниц, и доступные для свободного использования, можно найти в Интернете, в частности, на сайтах **http://freepsd.ws/** и **http://www.unishablon.com/**.

Логическая и физическая структуры Web-сайта

Осталось сказать несколько слов о структуре разрабатываемого нами сайта — как логической, так и физической.

Логическая структура сайта определяется принципами, устанавливаемыми используемым фреймворком — Laravel. Каждый раздел сайта представляется отдельным

контроллером, а каждая страница раздела — отдельным действием этого контроллера. В таком контроллере мы реализуем все необходимые операции: вывод списка позиций, вывод сведений об отдельной позиции, добавление, правку и удаление позиций.

Мы не станем создавать отдельный административный раздел сайта, предназначенный для работы с его внутренними данными. Мы просто будем в процессе генерирования страниц выводить на них гиперссылки, ведущие на страницы добавления, правки и удаления, и, разумеется, ограничим доступ к этим операциям, дав его только зарегистрированным пользователям.

В мобильной редакции сайта мы принципиально исключим все административные операции: редактирование списков категорий, подкатегорий, статей, комментариев и пользователей. Выполнять такие задачи на маленьких экранах планшетов и, тем более, смартфонов очень неудобно. Поскольку мы уже на стороне сервера можем определить, с какого устройства посетитель зашел на сайт, то реализуем это, просто блокировав операцию входа.

Что касается физической структуры сайта, то и в этом случае Laravel не оставляет нам особого выбора, уже при формировании нового сайта создавая почти все необходимые нам папки для хранения внешних таблиц стилей и файлов сценариев. Если же нам понадобятся папки для хранения каких-либо других данных, например, графических файлов с элементами оформления, мы без проблем создадим их.

«СЭП» — Web-сайт электронных публикаций

Итак, все подготовительные действия выполнены, дизайн сайта подобран, его структура — как логическая, так и физическая — сформированы. Настала пора приступать непосредственно к разработке.

Назовем наш новый сайт просто «СЭП» — это сокращение от фразы «сайт электронных публикаций».

Дизайн сайта

Поскольку это наш первый сайт, мы выберем для него относительно простой дизайн. Например, такой, какой показан на рис. 35.1.

Для его реализации мы применим рамочную разметку, описанную в *главе 13*. Она позволит нам добиться нужного результата быстро и с относительно небольшим объемом кода.

Создать эффект слабоконтрастного графического фона можно двумя путями. Первый: графическое изображение соответствующим образом обрабатывается в графическом редакторе и указывается в качестве фона самой страницы. Второй: в качестве фона страницы задается необработанное изображение, после чего на страницу помещается фиксированный элемент, занимающий всю ее площадь и имеющий полупрозрачный фон, все остальные элементы, создающие разметку, выводятся поверх этого. Этот-то путь мы и изберем в *главе 36*.

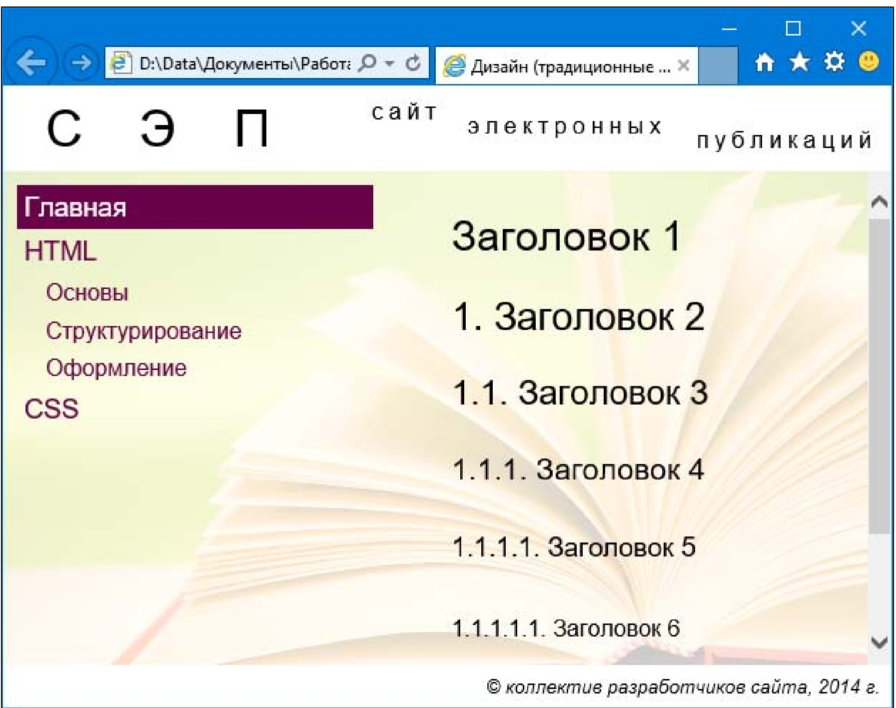


Рис. 35.1. Дизайн сайта электронных публикаций «СЭП»

Создание и настройка проекта

Сначала проверим, установили ли мы пакет хостинга OpenServer (как его установить, описано в *приложении 1*). Удалим все содержимое папки domains\localhost, расположенной в папке, где установлен этот пакет.

Запустим OpenServer, откроем входящую в состав этого пакета консоль и наберем в ней две команды, на забывая завершать каждую нажатием клавиши <Enter>:

```
cd domains
composer create-project laravel\laravel localhost --prefer-dist
```

Первая команда выполнит переход в папку domains. Вторая команда запустит процесс создания нового проекта и установку всех необходимых библиотек, включая сам Laravel, в папку localhost.

Сразу же установим библиотеку Captcha for Laravel, последовательно отдав в консоли команды:

```
cd localhost
composer require mews/captcha
```

Первая команда выполнит переход в папку localhost, вторая, собственно, установит саму библиотеку.

> ***На заметку***
>
> Имеет смысл также установить библиотеку doctrine/dbal, без которой нам, скорее всего, не получится выполнить миграции, изменяющие структуру таблиц в базе данных. Установим ее отдачей команды:
>
> ```
> composer require doctrine/dbal
> ```

Вынесем в папку resources\views\vendor шаблоны пагинатора и почтовых уведомлений, для чего отдадим команду:

```
php artisan vendor:publish
```

Далее выполним создание маршрутов и шаблонов для страниц регистрации, входа, запуска процедуры сброса пароля и выполнения его сброса. Это делается с помощью команды:

```
php artisan make:auth
```

Теперь создадим в любом простейшем текстовом редакторе файл с содержимым:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteRule ^(.*)$ public/$1 [L]
</IfModule>
```

И сохраним его в папке проекта под именем .htaccess.

В папке public найдем папки css и js, созданные при формировании нового проекта, и удалим все имеющиеся в них файлы. Все равно мы будем создавать представление и поведение нашего сайта самостоятельно.

Для хранения данных сайта мы создадим пустую базу под названием `epub`. Также мы занесем в список СУБД MySQL нового пользователя с именем `epub` и таким же паролем, который впоследствии, перед публикацией сайта, мы изменим на более стойкий. (Работа с базами данных MySQL описана в *приложении 2*.)

Откроем файл .env, хранящийся в самой папке проекта. Исправим настройки, касающиеся базы данных сайта, чтобы они выглядели следующим образом:

```
DB_DATABASE=epub
DB_USERNAME=epub
DB_PASSWORD=epub
```

Также исправим настройки, задающие параметры отправки почты. Собственно, нам следует изменить всего лишь одну настройку:

```
MAIL_DRIVER=mail
```

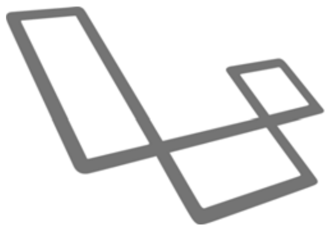
Сохраним файл .env и закроем его.

Откроем файл config\app.php и добавим в его код следующие строки (выделены полужирным шрифтом):

```
'providers' => [
  . . .
  Mews\Captcha\CaptchaServiceProvider::class,
]
```

```
. . .
'aliases' => [
  . . .
  'Captcha' => 'Mews\Captcha\Facades\Captcha',
]
```

Этим мы закончим установку библиотеки Captcha for Laravel. Исправив код, сохраним и закроем файл.



ГЛАВА 36

Создание дизайна Web-страниц

Ранее говорилось, что шаблоны, на основе которых будут генерироваться динамические Web-страницы, удобнее создавать, используя готовые статические страницы. Как правило, разработка сайта и начинается с подготовки этих страниц, включающих некоторое отладочное содержание, — так мы сразу можем в процессе разработки оценить их дизайн.

Особенности создания представления для Web-страниц

Прежде чем мы приступим собственно к разработке статических страниц, нужно сказать пару слов об их представлении. Это касается, по большей части, не самого оформления страниц, а технических вопросов, связанных с хранением стилей, задающих их представление.

Как мы знаем еще из *главы 7*, практически все стили хранятся во внешних таблицах стилей. Это позволяет нам без проблем задать для всех страниц сайта единое представление, просто привязав к ним нужные таблицы стилей.

Часто стили хранятся в нескольких внешних таблицах стилей. Это позволяет разделить стили по назначению.

- «Общая» таблица стилей хранит стили, действующие на все страницы обеих редакций сайта: и «компьютерной», и «мобильной». Хранящиеся в ней стили задают представление, одинаковое для всех страниц, — это могут быть цвета элементов и фона, параметры рамок и проч.

  Понятно, что эта таблица стилей привязывается ко всем страницам сайта без исключения.

- Отдельные таблицы стилей для «компьютерной» и «мобильной» редакций сайта. Они указывают размеры шрифтов, параметры разметки и др.

  Эти таблицы стилей привязываются к страницам какой-то одной редакции сайта: «компьютерной» или «мобильной».

- Таблицы стилей, указывающие представление, которое действует лишь при печати страниц. Как правило, их делают универсальными (разумеется, по возможности), подходящими для обеих редакций сайта.

Web-страницы для традиционных компьютеров

Начнем мы с разработки страниц, предназначенных для традиционных компьютеров. Сделаем для них разметку, отладочное содержание и представление.

Разметка

Откроем текстовый редактор и наберем в нем HTML-код, приведенный в листинге 36.1. Сохраним его в файле с «говорящим» именем pc.html.

Листинг 36.1

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="basic.css" type="text/css">
    <link rel="stylesheet" href="pc.css" type="text/css">
    <title>Дизайн (традиционные компьютеры)</title>
  </head>
  <body>
    <div id="container"></div>
    <header>
    </header>
    <nav>
    </nav>
    <section>
    </section>
    <footer>
    </footer>
  </body>
</html>
```

Здесь мы формируем элементы, составляющие разметку, пока без содержимого, и служебный блок `container`, с применением которого впоследствии будет создан эффект слабоконтрастного фона (как мы условились в *главе 35*). Также мы выполняем привязку обеих таблиц стилей: basic.css (общее представление) и pc.css (представление для «компьютерной» редакции сайта).

Что касается элементов, формирующих разметку страницы, то они создаются тегами семантической разметки HTML 5:

- `<header>`— «шапка» страницы;
- `<nav>` — панель навигации;
- `<section>` — блок основного содержания;
- `<footer>` — «поддон» страницы.

Ранее, до широкого распространения HTML 5, для этих целей применялись обычные блоки (теги `<div>`). Но мы, приверженцы современных интернет-технологий, задействуем их лишь для служебных целей.

Начальное представление

Начальное представление укажет основные параметры страницы: цвет текста и фона, название и размер шрифта различных элементов и др.

Сначала зададим начальное представление, общее для обеих редакций сайта. Создадим новый текстовый файл, введем в него код, приведенный в листинге 36.2, и сохраним под именем basic.css.

Листинг 36.2

```
body                        { color: #000000;
                              background-color: white;
                              font-family: sans-serif;
                              margin: 0px; }
h1, h2, h3, h4, h5, h6      { font-weight: normal; }
a:link, a:active, a:hover { color: #680148; }
a:visited                   { color: #ab86b6; }
```

Для самой страницы (тега `<body>`) мы указали черный цвет текста, белый цвет фона и шрифт без засечек (семейство sans-serif). Еще мы задали внешние отступы, равные нулю, — таким образом мы устранили просветы между границами страницы и ее содержанием.

Для всех тегов заголовков мы задали обычное начертание, а не используемое по умолчанию полужирное. Это придаст дизайну страниц «легкость».

Наконец, мы указали для обычных гиперссылок, активной гиперссылки и гиперссылки, на которую наведен курсор мыши, темно-красный цвет текста (RGB-код 680148), а для посещенных гиперссылок — светло-красный цвет (ab86b6). Эти параметры будут действовать на все гиперссылки страницы, за исключением пунктов панели навигации, которую мы сделаем потом.

> ***На заметку***
>
> Цвета для сайта можно подбирать самостоятельно, а можно выбрать из цветовой схемы, составленной профессиональным дизайнером. Набор бесплатных цветовых схем можно найти в Интернете, например, на сайте ColourLovers (**http://www.colourlovers.com/**).

Теперь создадим эффект слабоконтрастного графического фона. Он будет присутствовать только на страницах «компьютерной» редакции сайта, следовательно, соответствующие стили мы поместим во вновь созданную таблицу стилей pc.css. Код этих стилей можно увидеть в листинге 36.3.

Листинг 36.3

```
body {
  background: url(background.jpg) center top/cover no-repeat fixed;
}
div#container, header, nav, section, footer { position: fixed; }
div#container { top: 0px;
                right: 0px;
                bottom: 0px;
                left: 0px;
                background-color: rgba(255, 255, 255, 0.8); }
```

Здесь мы сначала указываем для самой страницы (ее секции тела) подходящее графическое изображение, которое можно без труда найти в Интернете (предполагается, что фоновое изображение сохранено в файле background.jpg.) Располагаем его по середине верхнего края страницы, делаем так, чтобы оно покрывало всю страницу без искажения пропорций, отключаем повторение и фиксируем его.

Далее мы превращаем блок `container`, а заодно и все элементы, составляющие разметку страницы, в фиксированные элементы. И, наконец, растягиваем блок `container` на всю страницу и задаем для него полупрозрачный белый фон.

«Шапка»

Поскольку, как говорилось ранее, это наш первый сайт, то и дизайн для него мы подберем попроще. К тому же автор книги не силен в разработке дизайна страниц (если понимать его как рисование графического макета), хотя и неплохо умеет делать страницы на основе уже готовых макетов.

Тем не менее, авторский, если так можно сказать, дизайн «шапки» отражает последние тенденции в построении таких элементов страницы — в частности, использование свободно позиционируемых блоков для размещения фрагментов текста. Так что читатели, по крайней мере, узнают, как делается современная «шапка».

Откроем страницу pc.html и вставим в тег `<header>` код, который создаст содержимое «шапки»:

```
<header>
  <div id="header_1"><h1>СЭП</h1></div>
  <div id="header_2"><h2>сайт</h2></div>
  <div id="header_3"><h2>электронных</h2></div>
  <div id="header_4"><h2>публикаций</h2></div>
</header>
```

Здесь мы создали четыре фрагмента текста, которые чуть позже расположим в различных местах «шапки». Поскольку свободно позиционировать мы можем только блоки и элементы семантической разметки, каждый из этих фрагментов помещен внутрь блока, для которого задано имя, — это имя мы используем в составе селектора будущего стиля, задающего местоположение фрагмента.

Хоть «шапка» у разных редакций сайта будет индивидуальная, часть параметров ее представления будет являться общей для них. Так что мы можем со спокойной совестью вынести в таблицу стилей basic.css стили, задающие общее представление:

```
header, footer       { background-color: white; }
header h1, header h2 { margin: 0px; }
```

Эти стили укажут для самого тега `<header>`, а заодно и для тега «поддона» `<footer>`, белый (`white`) цвет фона — так мы дополнительно выделим «шапку» на фоне остальных элементов страницы. А для вложенных в этот тег заголовков первого и второго уровней зададим нулевые внешние отступы, чтобы потом не испытывать проблемы с их позиционированием.

Теперь займемся кодом, задающим для «шапки» специфичное представление. Этот код мы добавим в таблицу стилей pc.css. Он достаточно сложен, так что мы рассмотрим его по частям.

```
header { top: 0px;
         right: 0px;
         left: 0px;
         height: 60px; }
```

Так мы поместим саму семантическую «шапку» (тег `<header>`) в верхней части страницы и укажем ей высоту 60 пикселов (это значение подобрано автором экспериментально).

```
header h1 { font-size: 36px;
            letter-spacing: 40px;
            text-transform: uppercase; }
header h2 { font-size: 16px;
            letter-spacing: 4px;
            text-transform: lowercase; }
```

Здесь мы задаем параметры для заголовков первого (текст «СЭП») и второго (остальные фрагменты текста) уровней: размер шрифта, дополнительное пространство между символами и их регистр.

```
header div { position: absolute; }
#header_1  { left: 30px;
             top: 10px; }
#header_2  { right: calc(2 * 100% / 3 - 100px);
             top: 10px; }
#header_3  { right: calc((2 * 100% / 3) / 2 - 50px);
             top: 20px; }
#header_4  { right: 10px;
             top: 30px; }
```

А здесь указываем для всех блоков, входящих в состав «шапки», свободное позиционирование и задаем их местоположение. Мы располагаем блоки так, чтобы они подстраивались под ширину окна Web-обозревателя (сами координаты блоков подобраны автором экспериментальным путем).

В результате мы должны получить «шапку», показанную на рис. 35.1. Да, она далеко не шедевр, но ведь мы еще только учимся...

Панель навигации

Поместим в тег семантической разметки `<nav>` код, показанный в листинге 36.4. Это отладочный код; он позволит нам оценить, как будет выглядеть наша панель навигации.

Листинг 36.4

```
<nav>
  <a class="active" href="#">Главная</a>
  <a href="#">HTML</a>
  <div>
    <a href="#">Основы</a>
    <a href="#">Структурирование</a>
    <a href="#">Оформление</a>
  </div>
  <a href="#">CSS</a>
</nav>
```

Обычно пункты панели навигации создаются с помощью блоков, в которые помещаются гиперссылки. Однако наш подход является более удачным. Пункты, представляющие категории, мы создадим непосредственно на основе гиперссылок, которые впоследствии с помощью особого стиля превратим в блочные элементы. А пункты, представляющие подкатегории, относящиеся к категории, мы поместим внутрь блока, следующего за пунктом-категорией, и также реализуем в виде гиперссылок.

Оформление панели навигации мы поместим в таблицу стилей pc.css, поскольку страницы «мобильной» редакции сайта вообще не будут содержать эту панель. Создающий это оформление код достаточно сложен, и мы рассмотрим его по частям.

```
nav { top: 70px;
       bottom: 34px;
       left: 10px;
       width: 270px;
       overflow: auto; }
```

Так мы помещаем панель навигации в левой части страницы, под «шапкой», задаем для нее ширину в 270 пикселов и вывод полос прокрутки в случае переполнения.

```
nav a { display: block; }
```

И все гиперссылки, находящиеся в панели навигации, превращаем в блочные элементы.

```
nav > a { font-size: 14pt;
          padding: 5px 5px;
          width: 240px; }
```

Этот стиль задаст оформление для гиперссылок, непосредственно вложенных в тег `<nav>`, т. е. представляющих категории. Здесь нам все знакомо и понятно, за исключением одного важного момента: ширину пунктов панели навигации мы задаем меньшей, чем ширина самой панели. Указанное нами значение 240 пикселов при добавлении величин внутренних отступов слева и справа (по 5 пикселов) даст 250 пикселов — в результате между правыми краями пунктов и правым краем панели навигации образуется просвет в 20 пикселов, в котором будет выводиться полоса прокрутки.

```
nav > div { width: 250px; }
```

У блока, который создаст своего рода подменю с пунктами-подкатегориями, указываем ширину в 250 пикселов, то есть равной полной ширине пунктов-категорий.

```
nav > div > a { font-size: 12pt;
                padding: 4px 5px 4px 20px;
                width: 225px; }
```

Для пунктов-подкатегорий мы указываем размер шрифта меньший, чем у пунктов-категорий, и увеличенный на 5 пикселов внутренний отступ слева. Соответственно, мы должны скорректировать для них значение ширины, которое составит 225 пикселов.

```
nav a:link, nav a:visited, nav a:active, nav a:hover {
  color: #680148;
  text-decoration: none;
}
```

Для любых гиперссылок, которые формируют пункты панели навигации, мы указываем одинаковый темно-вишневый цвет текста и отсутствие любых его «украшений» — так мы уберем у гиперссылок подчеркивание.

```
nav a:active, nav a:hover, nav a.active {
  color: white;
  background-color: #680148;
  transition: color 0.5s, background-color 0.5s;
}
```

Для гиперссылок панели навигации — активной, той, на которую наведен курсор мыши, и той, к которой привязан стилевой класс `active` (он укажет пункт, соответствующий текущей странице), — мы зададим белый цвет текста и темно-вишневый цвет фона («инверсный» вид). Также мы укажем анимацию этих параметров в течение 0,5 секунды.

```
nav a:link, nav a:visited { transition: color 1s, background-color 1s; }
```

А для обычных и посещенных гиперссылок мы задали анимацию тех же параметров, но уже в течение одной секунды. Это создаст красивый эффект «волны» при проведении курсора мыши по панели навигации.

Панель навигации, которая должна у нас получиться в результате, также можно увидеть на рис. 35.1.

Блок основного содержания

Настал черед блока основного содержания. Здесь нам придется повозиться, задавая представления для текста, заголовков, форм, элементов управления, изображений, а также аудио- и видеороликов, которые будут в нем присутствовать.

Параметры самого блока основного содержания

Но сначала зададим параметры самого этого блока — тега `<section>`, которым он создается. Их немного.

Хоть этот блок и будет различаться в разных редакциях сайта, часть параметров его представления будет общей. Указывающий их стиль мы занесем в таблицу стилей basic.css:

```
section { padding: 5px; }
```

Он установит внутренние отступы равными 5 пикселам.

Стиль, задающий специфическую для «компьютерной» редакции сайта часть представления, мы занесем в таблицу стилей pc.css:

```
section { top: 60px;
          bottom: 24px;
          left: 310px;
          right: 0px;
          overflow: auto; }
```

Здесь мы помещаем элемент основного содержания в правой части страницы, правее панели навигации, и указываем для него вывод полос прокрутки при переполнении.

Параметры текста

Параметры текста, выводящегося в блоке основного содержания, будут разниться от одной редакции сайта к другой. Поэтому мы поместим задающие их стили (они показаны в листинге 36.5) в таблицу стилей pc.css.

Листинг 36.5

```
p, div, pre, li, th, td { font-size: 10pt; }
h6                      { font-size: 12pt; }
h5                      { font-size: 14pt; }
```

```
h4                      { font-size: 16pt; }
h3                      { font-size: 18pt; }
h2                      { font-size: 20pt; }
h1                      { font-size: 22pt; }
a:link, a:visited       { text-decoration: none; }
a:active, a:hover       { text-decoration: underline; }
```

Здесь мы задаем размер шрифта, которым будет выведен текст абзацев, блоков, заголовков и ячеек таблиц, а также текст фиксированного форматирования. Еще мы задаем для обычных и посещенных гиперссылок отсутствие подчеркивания, а для активной гиперссылки и гиперссылки, на которую наведен курсор мыши, — его наличие.

Параметры нумерации заголовков

Посетителю будет много удобнее читать статью, особенно большую, если ее заголовки пронумерованы. А язык CSS предоставляет нам встроенные средства для включения в состав элементов страницы генерируемого содержания, в том числе нумерации (подробнее — в *главе 10*). Так почему бы ими не воспользоваться?

Давайте зарезервируем заголовки первого уровня для вывода названий статей и, соответственно, не будем их нумеровать. А для предварения отдельных частей этих статей оставим заголовки второго и последующего уровней.

Параметры автонумерации заголовков будут одинаковыми для обеих редакций сайта. Так что мы поместим стили, что их задают (листинг 36.6), в таблицу стилей basic.css.

Листинг 36.6

```
section {
  counter-reset: counter-h2 counter-h3 counter-h4 counter-h5 counter-h6;
}
section h2:before {
  counter-increment: counter-h2;
  counter-reset: counter-h3;
  content: counter(counter-h2) ". ";
}
section h3:before {
  counter-increment: counter-h3;
  counter-reset: counter-h4;
  content: counter(counter-h2) "." counter(counter-h3) ". ";
}
section h4:before {
  counter-increment: counter-h4;
  counter-reset: counter-h5;
  content: counter(counter-h2) "." counter(counter-h3) "."
  counter(counter-h4) ". ";
}
```

```
section h5:before {
  counter-increment: counter-h5;
  counter-reset: counter-h6;
  content: counter(counter-h2) "." counter(counter-h3) "."
  counter(counter-h4) "." counter(counter-h5) ". ";
}
section h6:before {
  counter-increment: counter-h6;
  content: counter(counter-h2) "." counter(counter-h3) "."
  counter(counter-h4) "." counter(counter-h5) "." counter(counter-h6)
  ". ";
}
```

Хоть код этих стилей велик, но лишь на первый взгляд кажется сложным. Мы можем сами разобраться, как он работает, поскольку все необходимые нам знания изложены в *главе 10*.

На рис. 36.1 можно увидеть заголовки различных уровней вложенности, пронумерованные согласно заданным нами параметрам. Отметим, что самый верхний заголовок, имеющий первый уровень вложенности, не пронумерован.

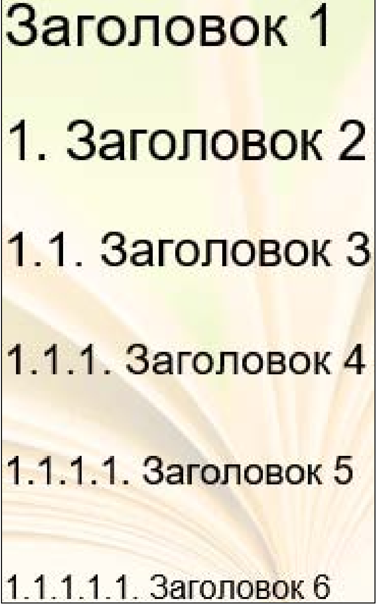


Рис. 36.1. Заголовки различного уровня вложенности с нумерацией

Параметры внедренных элементов

Ни одна приличная страница не обходится без внедренных элементов — по меньшей мере, графических изображений. И ни одна приличная статья не обойдется без иллюстраций — по меньшей мере, обычных рисунков.

Давайте условимся вот о чем:

- изображения, аудио- и видеофайлы мы будем помещать в теги `<figure>`;
- подписи к ним мы будем помещать в блоки с привязанным стилевым классом `sign`. Это заметно упростит нам преобразование тегов BBCode в соответствующий HTML-код (этим мы займемся в *главе 41*).

Вот пример HTML-кода, выводящего на страницу изображение с подписью, аудио- и видеоролик:

```
<figure>
  <img src="image_small.jpg">
</figure>
<div class="sign">Рисунок</div>
<figure><audio src="audio.mp3" controls></audio></figure>
<figure><video src="video.mp4" controls></video></figure>
```

Параметры внедренных элементов также будут общими для обеих редакций сайта. И мы поместим задающие их стили в таблицу стилей basic.css. Эти стили мы рассмотрим по очереди.

```
figure { text-align: center;
         margin: 40px 0px; }
```

Для тега `<figure>` мы здесь указываем выравнивание содержимого посередине и внешние отступы сверху и снизу в 40 пикселов, чтобы отделить иллюстрации от соседних элементов.

```
figure * { max-width: 100%;
           max-height: 300px; }
```

И для любых элементов, вложенных в тег `<figure>`, задаем максимальную ширину, равную ширине свободного пространства в родителе, и максимальную высоту в 300 пикселов. Так мы исключим ситуацию, когда слишком большой внедренный элемент занимает все окно Web-обозревателя, и одновременно дадим элементу возможность подстраивать свою ширину под ширину окна.

```
div.sign { font-style: italic;
           text-align: center;
           margin-top: -30px; }
```

И наконец, для подписей (блоков с привязанным стилевым классом `sign`) устанавливаем выравнивание посередине, курсивный текст и отрицательный внешний отступ сверху, чтобы приблизить подпись к расположенной выше иллюстрации.

На рис. 36.2 можно видеть изображение с подписью, аудио- и видеоролик, выведенные на страницу с учетом заданных нами параметров.

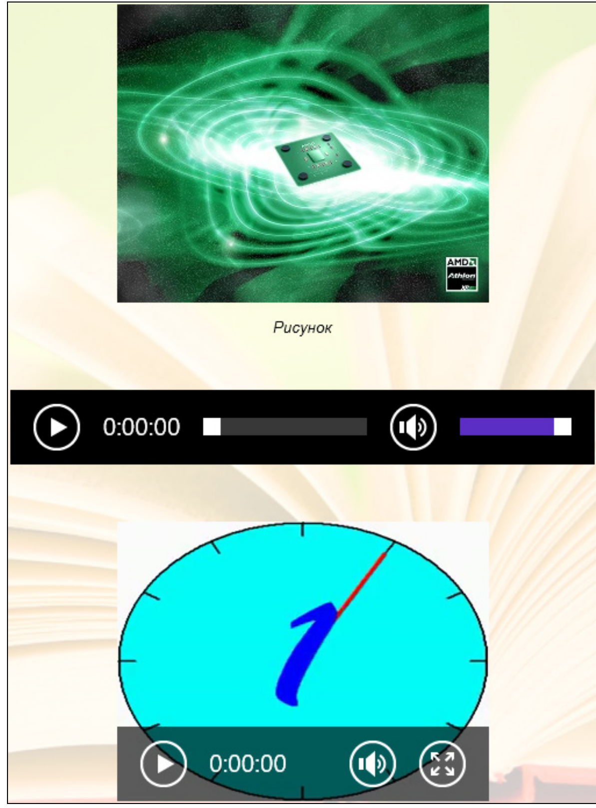


Рис. 36.2. Изображение с подписью, аудио- и видеоролик

Параметры Web-форм и элементов управления

Создавая формы и элементы управления, сначала мы условимся вот о чем:

- надписи к элементам управления мы будем создавать тегом `<label>`;
- надписи и элементы управления мы превратим в блочные элементы — это позволит нам без проблем и без лишнего HTML-кода расположить их друг под другом;
- кнопки мы выровняем по правому краю, чтобы сделать их более заметными.

В листинге 36.7 показан пример кода, создающего форму с полем ввода, областью редактирования, списком и кнопкой (как наиболее типичными элементами управления).

Листинг 36.7

```
<form action="#">
  <label>Поле ввода</label>
  <input type="text">
  <label>Область редактирования</label>
```

```
  <textarea></textarea>
  <label>Список</label>
  <select size="2">
    <option>Пункт 1</option>
    <option>Пункт 2</option>
  </select>
  <input type="submit" value="OK">
</form>
```

Бóльшая часть параметров форм и элементов управления будут общими для обеих редакций сайта. Стили, что их задают, мы поместим в таблицу стилей basic.css. Рассмотрим их по очереди.

```
form { background-color: white;
       padding: 5px;
       overflow: hidden; }
```

Здесь мы указываем для формы белый цвет фона, чтобы дополнительно выделить ее, и внутренние отступы.

Ранее мы условились, что выровняем кнопки по правому краю формы. Для этого мы превратим кнопки в плавающие элементы и сдвинем их к правому краю родителя. Чтобы родитель — форма — растягивалась по высоте, дабы вместить в себя все плавающие элементы, мы дополнительно добавили в приведенный ранее стиль атрибут `overflow` со значением `hidden`. (Об этом трюке рассказывалось в *главе 12*.)

```
label, input, textarea, select { display: block; }
```

Превращаем надписи и элементы управления в блочные элементы.

```
label             { margin: 10px 0px 2px 0px; }
label:first-child { margin-top: 0px; }
```

Для надписей задаем внешние отступы сверху и снизу, которые отделят надпись от предыдущего и следующего элементов управления. Для первой надписи в форме указываем нулевой отступ сверху, чтобы убрать лишний просвет между верхней границей формы и этой надписью.

```
input, textarea, select { padding: 5px;
                          border: 4px solid grey; }
```

Для всех элементов управления указываем серую (`grey`) сплошную рамку толщиной 4 пиксела — это сделает элементы управления заметнее. Также указываем внутренние отступы, чтобы визуально отделить текстовое содержимое элементов управления от их рамок.

```
textarea { font-family: monospace;
           width: calc(100% - 18px); }
```

Для областей редактирования дополнительно задаем два параметра. Первый — моноширинный шрифт (семейство monospace), таким шрифтом удобнее набирать текст статей. Второй — ширину, равную ширине свободного пространства в блоке

основного содержания за вычетом величин внутренних отступов слева и справа (по 5 пикселов) и толщины левой и правой сторон рамки (по 4 пиксела).

```
input[type=submit], input[type=button], input[type=reset] {
  float: right;
  color: white;
  background-color: grey;
  font-weight: bold;
  margin: 15px 0px 5px 0px;
  min-width: 80px;
  border: none;
}
```

Превращаем кнопки в плавающие элементы и сдвигаем их к правому краю родителя. Задаем для них белый цвет и полужирное начертание текста, серый цвет фона, внешние отступы, чтобы визуально отделить их от соседей, минимальную ширину в 80 пикселов и отсутствие рамки.

```
input ~ * { clear: both; }
```

И, наконец, для любого элемента, следующего за элементом управления, задаем вывод ниже всех плавающих элементов. Это нужно для того, чтобы любой элемент, следующий за кнопкой, был отображен под ней.

Осталось только создать стили, задающие для элементов управления анимационные эффекты. Их мы поместим в таблицу стилей pc.css, так как они нужны только для «компьютерной» редакции сайта.

```
input, textarea, select {
  font-size: 10pt;
  transition: color 1s, background-color 1s, border-color 1s;
}
```

Этот стиль создаст для элементов управления обратную анимацию. Анимированы будут цвета текста, фона и рамки, и анимация будет продолжаться одну секунду. Также он задаст размер шрифта, равный 10 пунктам.

```
input:focus, textarea:focus, select:focus, input:hover, textarea:hover,
select:hover { border-color: orange;
               transition: border-color 0.2s; }
```

Элементы управления при наведении курсора мыши и получении фокуса ввода сменят цвет рамки на оранжевый (`orange`) в течение 0,2 секунды.

```
input[type=submit], input[type=button], input[type=reset] {
  cursor: pointer;
}
```

При наведении курсора мыши на кнопку любого типа он сменит форму на «указующий перст».

```
input[type=submit]:focus, input[type=button]:focus,
input[type=reset]:focus, input[type=submit]:hover,
```

```
input[type=button]:hover, input[type=reset]:hover {
  color: black;
  background-color: orange;
  transition: color 0.2s, background-color 0.2s;
}
```

Также при наведении курсора мыши на кнопку она сменит цвет текста на черный, а цвет фона — на оранжевый в течение 0,2 секунды.

На рис. 36.3 представлена форма с полем ввода, областью редактирования, списком и кнопкой. Область редактирования имеет фокус ввода.

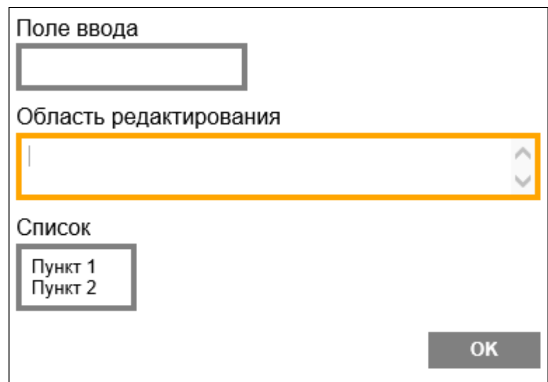


Рис. 36.3. Web-форма с различными элементами управления

«Поддон»

Осталось лишь сделать «поддон» страницы. Найдем в коде страницы pc.html тег `<footer>`, который его формирует, и вставим в него следующий небольшой код:

```
<footer>
  <p>© коллектив разработчиков сайта, 2017 г.</p>
</footer>
```

«Поддон» будет индивидуальным у разных редакций сайта, однако, как и в случае с «шапкой» и основным содержанием, часть его представления будет общей. Задающий его стиль мы добавим в таблицу стилей basic.css:

```
footer p { font-style: italic;
           text-align: right;
           margin: 0px; }
```

Здесь все нам уже знакомо. Ранее, делая представление для «шапки», мы указали для «поддона» еще и белый цвет фона.

Специфичным для «компьютерной» редакции стилям самое место в таблице стилей pc.css. Вот их код:

```
footer   { bottom: 0px;
           left: 0px;
           right: 0px;
```

```
          height: 14px;
          padding: 8px; }
footer p { font-size: 10pt; }
```

Тем самым «поддон», ранее превращенный нами в фиксированный элемент, мы поместили в самом низу страницы.

Web-страницы для мобильных устройств

Со страницами редакции сайта, предназначенной для мобильных устройств, будет легче — все общие для обеих редакций сайта параметры мы уже указали, а разметка у «мобильных» страниц организуется существенно проще.

Разметка

Откроем текстовый редактор и наберем в нем HTML-код, приведенный в листинге 36.8. Сохраним его в файле mobile.html.

Листинг 36.8

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="basic.css" type="text/css">
    <link rel="stylesheet" href="mobile.css" type="text/css">
    <title>Дизайн (мобильные устройства)</title>
  </head>
  <body>
    <header>
    </header>
    <section>
    </section>
    <footer>
    </footer>
  </body>
</html>
```

Сразу отметим, что панель навигации (тег `<nav>`) здесь отсутствует. Следовательно, нам необходимо предусмотреть альтернативный способ навигации по сайту, и мы это обязательно сделаем.

«Шапка»

«Шапка» страниц мобильной редакции сайта будет предельно простой. Мы сделаем ее в виде обычной строки текста:

```
. . .
<header>
  <h1>СЭП: сайт электронных публикаций</h1>
</header>
. . .
```

И сразу же введем во вновь созданную таблицу стилей mobile.css стили, которые зададут ее оформление:

```
header, footer { background-color: lightgrey; }
header         { padding: 4px; }
header h1      { font-size: 16px;
                 white-space: nowrap; }
```

Здесь для «шапки» и «поддона» мы задали светло-серый (`lightgrey`) цвет фона. Еще мы указали Web-обозревателю не разбивать текст на строки — если этого не сделать, на очень маленьких экранах надпись в «шапке» может оказаться разорванной на несколько строк, что испортит весь дизайн.

Блок основного содержания

Чтобы проверить дизайн основного содержания, мы можем вставить в тег `<section>` содержимое этого же тега, взятое из кода страницы pc.html.

Код стилей, задающих представление для содержимого этого блока, показан в листинге 36.9. Он также вводится в таблицу стилей mobile.css.

Листинг 36.9

```
p, div, pre, li, th, td, input, textarea, select {
  font-size: 12pt;
}
h6  { font-size: 14pt; }
h5  { font-size: 16pt; }
h4  { font-size: 18pt; }
h3  { font-size: 20pt; }
h2  { font-size: 22pt; }
h1  { font-size: 24pt; }
pre { font-size: 10pt;
      white-space: pre-wrap; }
a:link, a:visited, a:active, a:hover { text-decoration: underline; }
```

Здесь мы задали увеличенные по сравнению с «компьютерной» редакцией сайта размеры шрифтов. Помимо этого, мы указали:

- разрывать на строки текст фиксированного форматирования. Это позволит нам по мере возможности исключить горизонтальную прокрутку страниц, поскольку строки программного кода, которые обычно набираются подобным текстом, могут быть очень длинными;
- подчеркивать все гиперссылки без исключения.

«Поддон»

«Поддон» мы оставим без изменения — таким же, как и в «компьютерной» редакции страницы:

```
. . .
<footer>
  <p>© коллектив разработчиков сайта, 2017 г.</p>
</footer>
. . .
```

Мы изменим лишь его представление, занеся в таблицу стилей mobile.css стили:

```
footer   { padding: 4px; }
footer p { font-size: 9pt;
           white-space: nowrap; }
```

Для абзаца, вложенного в тег `<footer>`, мы укажем запрет на разрыв строк, как и для текста «шапки».

Страница «мобильной» редакции сайта в том виде, в каком она будет выведена на экран, показана на рис. 36.4.

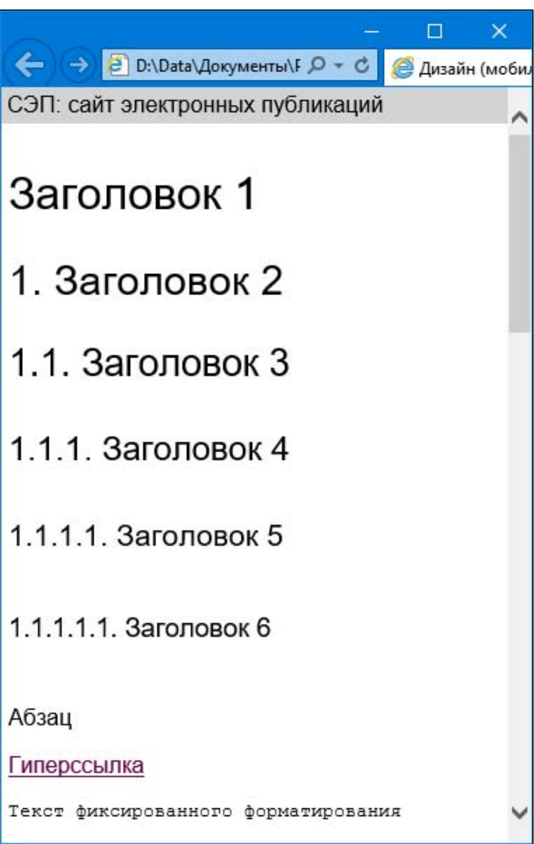


Рис. 36.4. Страница, предназначенная для мобильных устройств

Печатная редакция Web-сайта

Нам осталось лишь подготовить представление для печатной редакции страниц нашего сайта. Сделаем ее, создав специальную таблицу стилей и использовав для ее привязки соответствующий медиазапрос.

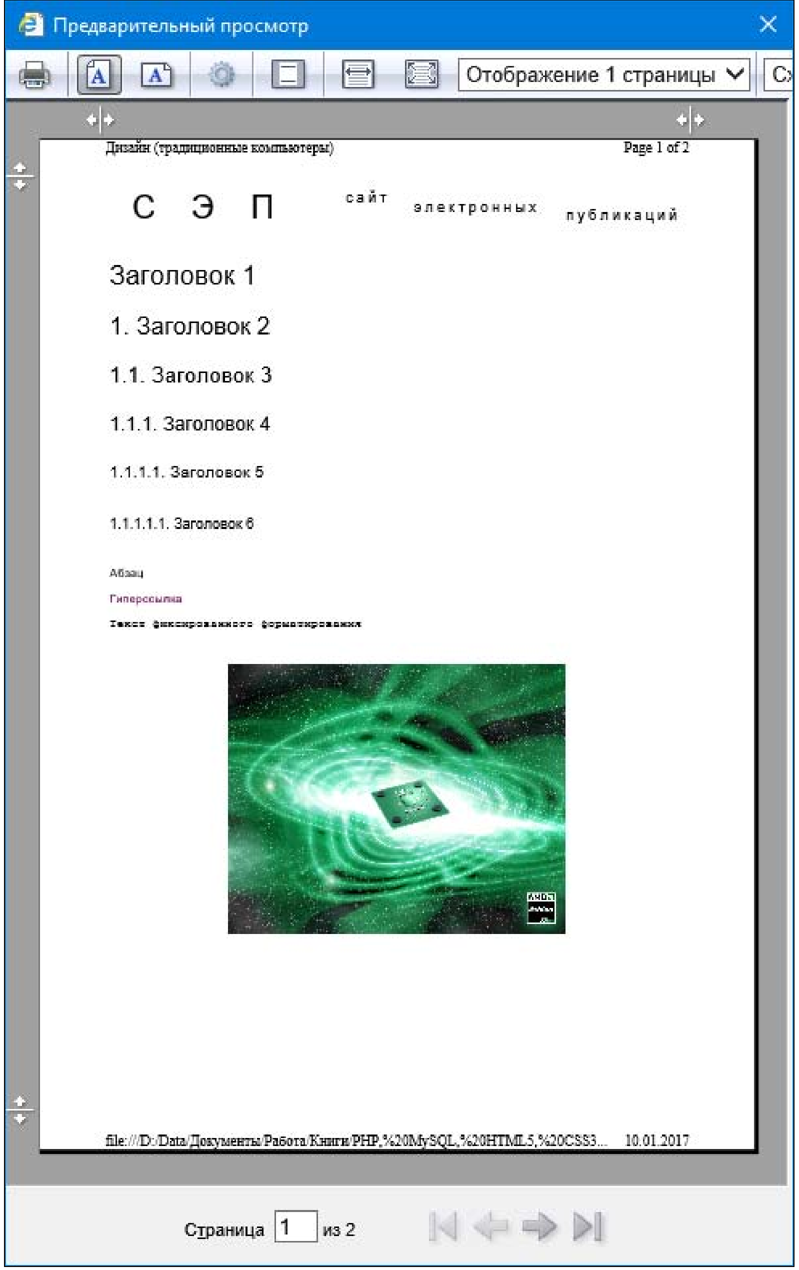


Рис. 36.5. Печатная редакция страницы

Таблицу стилей, что задаст представление печатной редакции страниц, мы назовем print.css. Ее код совсем невелик:

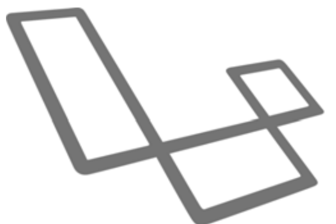
```
nav, audio, video      { display: none; }
header, section, footer { position: static; }
```

Здесь мы просто-напросто скрываем панель навигации и превращаем «шапку», основное содержание и «поддон» в непозиционируемые элементы. В результате на печатной странице будут выведены лишь три последних элемента разметки — один за другим. Также мы скроем аудио- и видеоролики — все равно на печать они не выводятся.

Получившаяся у нас таблица стилей является универсальной — она подойдет для обеих редакций сайта. Для ее привязки к страницам мы используем такой тег:

```
<link rel="stylesheet" href="print.css" type="text/css" media="print">
```

Печатную редакцию страницы, выведенную в окне предварительного просмотра Microsoft Internet Explorer, можно увидеть на рис. 36.5.



ГЛАВА 37

Интерактивные элементы

Формально программирование интерактивных элементов относится к созданию дизайна страниц. Однако это процесс, с одной стороны, достаточно сложный, а с другой, требующий написания объемистых Web-сценариев, поэтому он и описывается в отдельной главе.

Мы сделаем спойлер и лайтбокс, которые можно будет использовать в тексте статей, и блокнот с вкладками, на основе которого потом создадим интерфейс хранилища файлов. Все нужные знания мы получили в *части I* этой книги.

Спойлер

Спойлер, или *раскрывающаяся панель*, — один из популярнейших сейчас интерактивных элементов общего назначения. В изначальном состоянии он выглядит как небольшая полоска с заголовком, при щелчке на котором полоска разворачивается и открывает содержимое спойлера. Повторный щелчок на заголовке сворачивает его.

Формирование спойлера

Решим, с помощью каких элементов мы станем создавать на странице спойлеры.

- Сам спойлер будет представлять собой тег семантического примечания `<aside>`.
- Первым потомком этого блока будет гиперссылка, которую мы с помощью особого стиля превратим в блочный элемент и которая получит в качестве содержимого текст **Спойлер**. Щелчок на ней вызовет выполнение сценария, разворачивающего или сворачивающего спойлер. Применение гиперссылки позволит нам дать возможность посетителю разворачивать и сворачивать спойлер с клавиатуры.
- В начале текста гиперссылки будет присутствовать встроенный контейнер (тег `<span>`) с последовательностью символов "[+]" — эти символы сформируют индикатор, сообщающий о состоянии спойлера: свернут он или развернут. Знак "+" будет говорить о том, что спойлер свернут, и приглашать посетителя развернуть

его, знак "–" — наоборот. Индикатор мы создадим с помощью генерируемого содержания — это позволит нам несколько упростить программный код.

- За заголовком спойлера поместится блок, который вместит в себя содержимое спойлера, раскрывающееся при щелчке на заголовке.
- Сворачивание спойлера мы будем выполнять привязкой к тегу `<aside>`, формирующему сам спойлер, стилевого класса `condensed`.

Пример HTML-кода, создающего спойлер согласно определенным нами правилам, показан в листинге 37.1.

Листинг 37.1

```
<aside>
  <a href="#"><span></span>Спойлер</a>
  <div>
    <p>Это содержимое спойлера.</p>
    <figure><img src="image_small.jpg"></figure>
  </div>
</aside>
```

В качестве содержимого спойлера здесь используются абзац и изображение, оформленное согласно правилам, которые мы установили в *главе 36*.

Представление спойлера

Представление спойлера будет состоять из довольно большого количества стилей, которые мы разнесем по разным таблицам стилей. Начнем с представления, которое будет общим для обеих редакций нашего сайта: «компьютерной» и мобильной, — задающие его стили мы поместим в таблицу стилей basic.css. Все эти стили описаны далее.

- Сам блок спойлера будет у нас обведен рамкой — это обычное оформление подобного рода элементов:

```
aside { margin-top: 20px;
        border: 1px solid grey; }
```

- Гиперссылку — заголовок спойлера — мы превратим в блочный элемент и зададим для нее внутренние отступы, чтобы текст не налезал на ее границу:

```
aside > a { display: block;
            padding: 2px 5px; }
```

- Для гиперссылки заголовка укажем отсутствие подчеркивания и, в обычном состоянии, белый цвет текста и серый (как и у рамки) цвет фона — так мы визуально выделим заголовок спойлера:

```
aside > a:link, aside > a:visited, aside > a:active, aside > a:hover {
  text-decoration: none;
}
```

```
aside > a:link, aside > a:visited {
  color: white;
  background-color: grey;
  transition: color 1s, background-color 1s;
}
```

- Для индикатора состояния спойлера мы зададим моноширинный шрифт — так он будет выглядеть аккуратнее:

```
aside > a span { font-family: monospace; }
```

- Зададим текст индикатора, обозначающий развернутое состояние спойлера:

```
aside > a span:before { content: "[-] "; }
```

- Для блока содержимого указываем скрытие содержимого, которое выходит за пределы блока (иначе оно будет выводиться на экран, даже если спойлер свернут), внутренние отступы и продолжительность анимации, в которую будут вовлечены высота и внутренние отступы элемента:

```
aside > div { padding: 10px;
              overflow: hidden;
              transition: height 0.5s, padding 0.5s; }
```

- Многие элементы, в частности, абзацы и заголовки, по умолчанию выводятся со значительным внешним отступом сверху, что в совокупности с внутренним отступом самого блока содержимого, заданного нами ранее, может дать очень большой и очень некрасивый просвет. Поэтому для первого потомка блока содержимого, который может быть любым тегом, указываем отсутствие внешнего отступа сверху:

```
aside > div *:first-child {     margin-top: 0px; }
```

- Для свернутого спойлера обязательно задаем величины внутренних отступов сверху и внизу равными нулю. В противном случае высота свернутого спойлера будет не нулевой, а равной сумме значений этих отступов:

```
aside.condensed > div { padding: 0px 10px;
                        transition: height 0.5s, padding 0.5s; }
```

- Укажем текст индикатора, обозначающего свернутое состояние спойлера:

```
aside.condensed > a span:before { content: "[+] "; }
```

Теперь зададим представление спойлера, специфичное для «компьютерной» редакции страниц. Оно включает два стиля, который мы поместим в таблицу стилей pc.css:

- задаем продолжительность обратной анимации, в которую будут вовлечены цвета текста и фона гиперссылки, создающей заголовок спойлера (прямую анимацию мы создадим чуть позже):

```
aside > a:link, aside > a:visited {
  transition: color 1s, background-color 1s;
}
```

- указываем для гиперссылки заголовка при наведении курсора мыши черный цвет текста на оранжевом фоне. Также задаем продолжительность прямой анимации, в которую будут вовлечены эти параметры:

```
aside > a:active, aside > a:hover {
  color: black;
  background-color: orange;
  transition: color 0.2s, background-color 0.2s;
}
```

Программирование спойлера

Код сценария, который «вдохнет жизнь» в наши спойлеры, мы поместим в файл сценария main.js. И не забудем привязать этот файл к обеим страницам: pc.html и mobile.html, — вставив в их секции заголовка такой тег:

```
<script src="main.js"></script>
```

Все сценарии, сохраненные в файле main.js, мы поместим в тело обработчика события `load` окна Web-обозревателя:

```
window.addEventListener("load", function() {
  // Сценарии
});
```

Так мы будем уверены, что весь необходимый код будет выполнен лишь после окончания загрузки страницы.

Программный код, который мы напишем, можно разделить на две части: сценарий, который будет выполнен сразу же по окончании загрузки страницы и произведет инициализацию спойлеров, и обработчик события щелчка на заголовке спойлера, который будет разворачивать и сворачивать его.

Сценарий инициализации спойлера показан в листинге 37.2. (Не забываем, что он, как и весь остальной код, должен находиться в теле функции-обработчика события `load` окна.)

Листинг 37.2

```
var arrSpoilers = document.querySelectorAll("section aside");
var el, a;
for (var i = 0; i < arrSpoilers.length; i++) {
  el = arrSpoilers[i];
  a = el.querySelector("a");
  a.addEventListener("click", spoilerHeaderClick);
  a = el.querySelector("div");
  el.fullHeight = a.clientHeight;
  a.style.height = "0px";
  el.className = "condensed"
}
```

Сначала мы ищем в элементе основного содержания (теге `<section>`) все семантические примечания (`<aside>`), т. е. все спойлеры. В цикле мы перебираем найденные элементы, привязываем к гиперссылкам в их заголовках обработчик события `click` — функцию `spoilerHeaderClick` (это вторая часть программного кода спойлера, и мы напишем ее чуть позже), сохраняем значение текущей высоты блока с содержимым спойлера, задаем для него нулевую высоту и привязываем к спойлеру стилевой класс `condensed`. В результате все спойлеры у нас изначально будут свернутыми.

Здесь мы вплотную столкнулись с проблемой, описанной в *главе 14* и связанной с некоторыми особенностями анимации в стиле CSS 3. Дело в том, что для атрибута стиля, вовлекаемого в процесс анимации, оба граничных значения должны быть заданы в абсолютных единицах измерения. Но мы не знаем точного значения высоты содержимого спойлера в развернутом состоянии, следовательно, должны его вычислить.

Сделать это не составляет проблемы — достаточно обратиться к свойству `clientHeight`, поддерживаемому всеми элементами страницы и описанному в *главе 17*. А для хранения высоты мы создадим в теге `<aside>`, формирующем спойлер, добавленное свойство `fullHeight` (подробнее о добавленных свойствах говорилось в *главе 16*).

В листинге 37.3 представлена вторая часть программного кода, задающего поведение спойлера. Это обработчик события щелчка мышью на его заголовке — функция `spoilerHeaderClick`. Мы поместим ее над кодом, выполняющим инициализацию спойлеров (см. листинг 37.2).

Листинг 37.3

```
function spoilerHeaderClick(evt) {
  var el = evt.target.parentElement;
  var oSB = el.querySelector("div");
  if (el.className == "condensed") {
    el.className = "";
    oSB.style.height = el.fullHeight + "px";
  } else {
    el.className = "condensed";
    oSB.style.height = "0px";
  }
}
```

Здесь мы получаем элемент-родитель гиперссылки, создающей заголовок спойлера, и этим родителем будет элемент семантического примечания, то есть сам спойлер. Далее мы добираемся до блока, хранящего его содержимое, и выполняем проверку, привязан ли к нему стилевой класс `condensed`, т. е. свернут ли спойлер.

- Если спойлер свернут, мы разворачиваем его, для чего убираем привязку к самому спойлеру стилевого класса `condensed` и задаем для блока содержимого значение высоты, сохраненное ранее в добавленном свойстве `fullHeight`.

- Если спойлер развернут, мы сворачиваем его, привязав к спойлеру упомянутый стилевой класс и указав для блока содержимого нулевую высоту.

На рис. 37.1 показаны два спойлера: один в свернутом, а другой в развернутом состоянии.

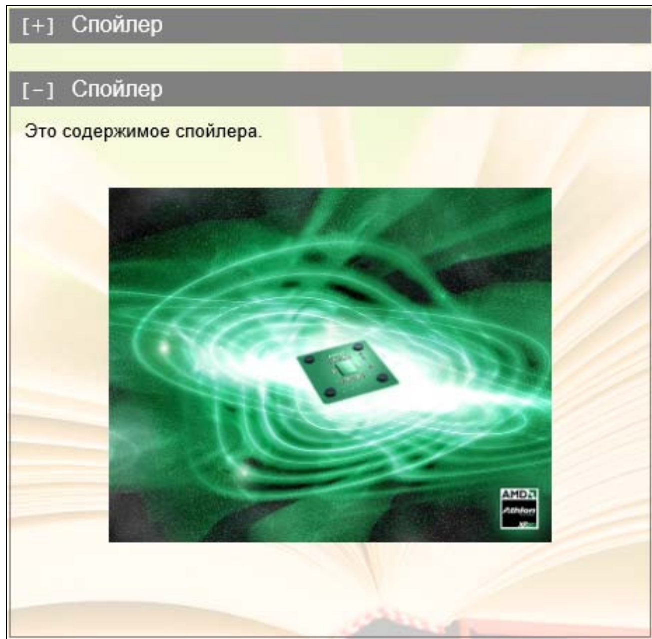


Рис. 37.1. Вверху — свернутый спойлер, а внизу — развернутый

Лайтбокс

Лайтбокс — это панель, выводящаяся поверх содержания страницы, показывающая какое-либо дополнительное содержание и имитирующая обычное окно Windows.

Очень часто лайтбокс применяется для вывода полноразмерной версии графического изображения, слишком большого, чтобы вывести его на странице непосредственно. В таком случае на странице помещается миниатюра этого изображения, при щелчке на которой разворачивается лайтбокс с основным изображением.

Формирование лайтбокса

Лайтбоксы мы станем создавать следующим образом:

- сам лайтбокс будет представлять собой семантическую иллюстрацию (`<figure>`) с привязанным стилевым классом `lightbox`;
- внутри лайтбокса будет помещаться гиперссылка, на которой посетитель будет щелкать мышью, чтобы развернуть лайтбокс:

- в этой гиперссылке будут находиться следующие элементы:
  - встроенный контейнер (<span>) с надписью **Щелкните, чтобы увеличить**;
  - графическое изображение (<img>) с миниатюрой, которая будет выводиться на странице;
  - графическое изображение с привязанным стилевым классом full-image с основным изображением, которое будет выводиться в лайтбоксе.

Пример HTML-кода, формирующего лайтбокс согласно заданным нами правилам, приведен в листинге 37.4.

Листинг 37.4

```
<figure class="lightbox">
  <a href="#">
    <span>Щелкните, чтобы увеличить</span><br>
    <img src="image_small.jpg">
    <img class="full-image" src="image.jpg">
  </a>
</figure>
```

Для успешного функционирования лайтбокса нам придется создать на странице еще два элемента:

- блок с именем lightbox_cover. Он будет выведен на экран при открытии лайтбокса, займет все пространство клиентской области окна Web-обозревателя и скроет содержание страницы, создав тем самым некое подобие фона для самого лайтбокса. При щелчке на этом блоке лайтбокс будет закрыт;
- блок с именем lightbox_window, представляющий собой сам лайтбокс. Он будет выведен поверх блока lightbox_cover и включит в свой состав тег <img>, указывающий на основное изображение, и свободно позиционируемый блок lightbox_close, содержимым которого станет гиперссылка с символом "X", имитирующая кнопку закрытия обычного окна Windows.

Изначально оба этих блока будут скрыты. Чтобы сделать их видимыми, мы привяжем к ним стилевой класс active. Создаст их написанный нами сценарий, который выполнит инициализацию лайтбоксов.

Представление лайтбокса

Почти все стили, задающие представление лайтбокса, будут одинаковыми для обеих редакций сайта. Их мы поместим в таблицу стилей basic.css и, как обычно, рассмотрим по очереди.

- У гиперссылок с текстом **Щелкните, чтобы увеличить**, входящих в состав лайтбокса, и у гиперссылки «кнопка закрытия» убираем подчеркивание и задаем серый цвет текста:

```
.lightbox a:link, .lightbox a:visited, .lightbox a:active,
.lightbox a:hover, #lightbox_close a:link, #lightbox_close a:visited,
#lightbox_close a:active, #lightbox_close a:hover {
  text-decoration: none;
}
.lightbox a:link, .lightbox a:visited, #lightbox_close a:link,
#lightbox_close a:visited {
  color: grey;
}
```

- Для текста **Щелкните, чтобы увеличить**, выводящегося над миниатюрой, указываем уменьшенный размер шрифта:

```
.lightbox a span { font-size: smaller; }
```

- Скрываем все основные изображения:

```
.lightbox a img.full-image { display: none; }
```

- Для блока `lightbox_cover` задаем серый цвет фона, полную прозрачность, невидимость, фиксированное позиционирование, размеры, совпадающие с размерами клиентской области окна Web-обозревателя, и курсор в виде «указующего перста»:

```
#lightbox_cover { background-color: grey;
                  opacity: 0.0;
                  position: fixed;
                  left: 0px;
                  top: 0px;
                  right: 0px;
                  bottom: 0px;
                  visibility: hidden;
                  cursor: pointer;
                  transition: opacity 0.5s; }
```

- Блок `lightbox_cover` с привязанным стилевым классом `active` станет видимым и полупрозрачным:

```
#lightbox_cover.active { opacity: 0.7;
                         visibility: visible;
                         transition: opacity 0.5s; }
```

- Блок `lightbox_window` получит белый цвет фона, довольно толстую (3 пиксела) черную рамку со скругленными углами, внутренние отступы в 20 пикселов со всех сторон и фиксированное позиционирование с отступами в 50 пикселов от краев клиентской области. Изначально он будет полностью прозрачным и невидимым.

  Также мы задали для содержимого этого блока гибкую разметку (см. *главу 13*). Это позволит нам расположить основное изображение по его центру максимально простым способом:

```
#lightbox_window { display: flex;
                   background-color: white;
                   padding: 20px;
                   border: 3px solid black;
                   border-radius: 10px;
                   position: fixed;
                   left: 50px;
                   top: 50px;
                   right: 50px;
                   bottom: 50px;
                   opacity: 0.0;
                   visibility: hidden;
                   transition: opacity 0.5s; }
```

- Тот же блок с привязанным стилевым классом `active` станет видимым и полностью непрозрачным:

```
#lightbox_window.active { opacity: 1.0;
                          visibility: visible;
                          transition: opacity 0.5s; }
```

- Для тега `<img>`, в котором будет выводиться основное изображение, указываем такие размеры, чтобы оно не вылезало за границы лайтбокса. Также указываем атрибут стиля `margin` со значением `auto`, что, вкупе с указанной для блока `lightbox_window` гибкой разметкой, расположит изображение в середине блока (этот трюк также описывался в *главе 13*):

```
#lightbox_window img { max-width: calc(100vw - 140px);
                       max-height: calc(100vh - 140px);
                       margin: auto; }
```

- Делаем блок `lightbox_close` свободно позиционируемым, помещаем его в правый верхний угол лайтбокса и выводим его содержимое — символ "X" — увеличенным полужирным шрифтом, чтобы он был лучше заметен:

```
#lightbox_close { font-size: larger;
                  font-weight: bold;
                  position: absolute;
                  top: 14px;
                  right: 14px; }
```

Стиль, специфичный для «компьютерной» редакции сайта, мы поместим в таблицу стилей pc.css. У гиперссылок с текстом **Щелкните, чтобы увеличить**, входящих в состав лайтбокса, и у гиперссылки «кнопка закрытия» при наведении курсора мыши указываем темно-красный цвет текста:

```
.lightbox a:active, .lightbox a:hover, #lightbox_close a:active,
#lightbox_close a:hover {
  color: #680148;
}
```

Программирование лайтбокса

Сценарий, задающий поведение лайтбоксов, можно разделить на три части: код инициализации, код, обрабатывающий событие щелчка мышью на миниатюре и собственно открывающий лайтбокс, и код, закрывающий лайтбокс при щелчке на элементах `lightbox_cover` и `lightbox_close`. Все три сценария мы поместим в тело обработчика события `load` окна, созданного ранее в файле сценария main.js.

В листинге 37.5 представлен код инициализации всех лайтбоксов, присутствующих на странице.

Листинг 37.5

```
var arrLightboxes = document.querySelectorAll↳
("section figure.lightbox a");
var cover, wnd, el2;
if (arrLightboxes.length > 0) {
  cover = document.createElement("div");
  cover.id = "lightbox_cover"
  document.body.appendChild(cover);
  wnd = document.createElement("div");
  wnd.id = "lightbox_window"
  wnd.innerHTML = "<div id='lightbox_close'><a href='#'>X</a></div>↳
  <img src=''>";
  document.body.appendChild(wnd);
  cover.addEventListener("click", lightboxCloseClick);
  el = document.getElementById("lightbox_close");
  el.addEventListener("click", lightboxCloseClick);
  for (var i = 0; i < arrLightboxes.length; i++)
    arrLightboxes[i].addEventListener("click", lightboxClick);
}
```

Хоть этот код весьма велик, но на самом деле он не так уж и сложен. Мы всего лишь получаем все семантические иллюстрации с привязанным стилевым классом `lightbox`, т. е. лайтбоксы, и, если хоть один из них присутствует на странице, создаем блоки `lightbox_cover` и `lightbox_window` со всем содержимым (в числе содержимого последнего — блок `lightbox_close`) и выполняем привязку обработчиков событий.

Отметим, что блоки `lightbox_cover` и `lightbox_window` мы сохранили в переменных `cover` и `wnd` соответственно, чтобы впоследствии без проблем получить к ним доступ, не выполняя поиск по именам.

Листинг 37.6 содержит код обработчика события щелчка мышью на миниатюре, который откроет лайтбокс.

Листинг 37.6

```
function lightboxClick(evt) {
  var el = evt.currentTarget;
  var h = el.querySelector("img.full-image").src;
  el = wnd.querySelector("img");
  el.src = h;
  cover.className = "active";
  wnd.className = "active";
}
```

Рис. 37.2. Миниатюра лайтбокса

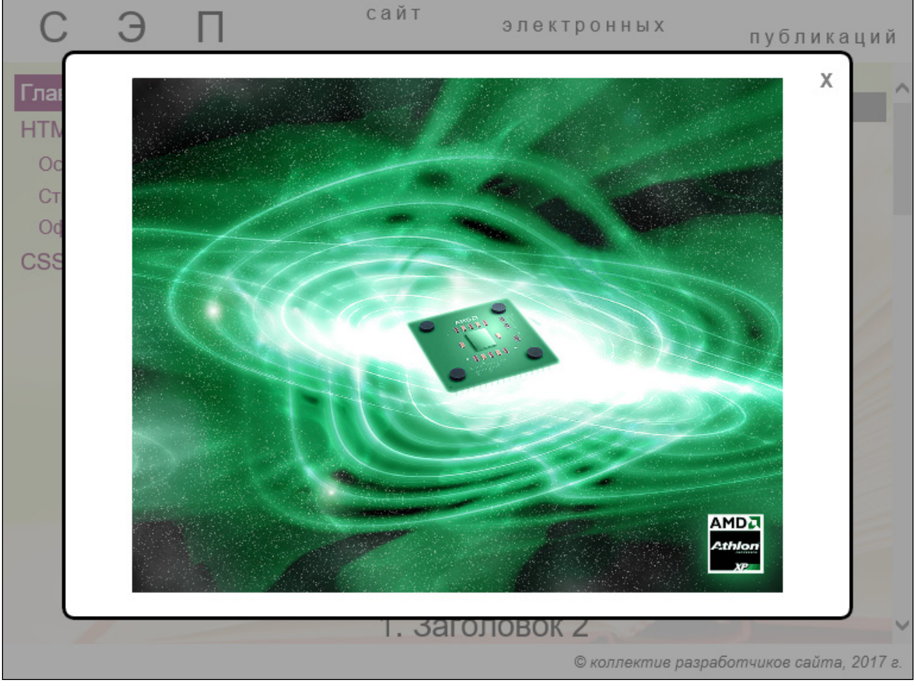


Рис. 37.3. Лайтбокс в развернутом виде

Здесь мы получаем интернет-адрес основного изображения, заносим его в тег `<img>`, входящий в состав блока `lightbox_window`, и выводим блоки `lightbox_cover` и `lightbox_window` на экран, привязав к ним стилевой класс `active`.

Осталось лишь написать код, который будет закрывать лайтбокс. Он совсем невелик — мы убираем привязку стилевого класса `active` к блокам `lightbox_cover` и `lightbox_window`, тем самым скрыв их:

```
function lightboxCloseClick() {
  cover.className = "";
  wnd.className = "";
}
```

На рис. 37.2 показана миниатюра лайтбокса с поясняющим текстом сверху. А рис. 37.3 представляет развернутый лайтбокс.

Блокнот

Блокнот понадобится нам, когда мы будем создавать интерфейс хранилища выгруженных пользователем файлов. На отдельных его страницах мы поместим списки файлов различных типов: изображения, аудио- и видеоролики и архивы, а также формы для их выгрузки.

Формирование блокнота

Сначала, как обычно, определимся, какие теги будут формировать блокнот.

- Набор корешков для переключения между вкладками блокнота будет помещен в блок с привязанным стилевым классом `tab-header`.
- Сами корешки будут представлять собой гиперссылки с именами вида *<имя блокнота>*`_header`*<номер вкладки от 0 до 9>*. (Десяти вкладок на блокнот вполне хватит для наших нужд.) Эти гиперссылки мы превратим в блочные элементы, сделаем плавающими и сдвинем к левому краю родителя.
- Чтобы активизировать корешок вкладки, мы привяжем к нему стилевой класс `active`.
- Набор самих вкладок будет помещен в блок с привязанным стилевым классом `tab-content`, который должен следовать сразу за блоком корешков.
- Отдельная вкладка блокнота должна представлять собой блок с привязанным стилевым классом `tab` и именем вида *<имя блокнота>*`_content`*<номер вкладки от 0 до 9>*. Содержимое вкладки должно находиться внутри этого блока.
- Чтобы активизировать вкладку блокнота, к формирующему ее блоку мы дополнительно привяжем стилевой класс `tab-visible`.

HTML-код, создающий блокнот, показан в листинге 37.7.

Листинг 37.7

```
<div class="tab-header">
  <a href="#" id="tab_header1" class="active">Вкладка 1</a>
  <a href="#" id="tab_header2">Вкладка 2</a>
  <a href="#" id="tab_header3">Вкладка 3</a>
  <a href="#" id="tab_header4">Вкладка 4</a>
</div>
<div class="tab-content">
  <div id="tab_content1" class="tab tab-visible">
    <!-- Содержимое вкладки 1 -->
  </div>
  <div id="tab_content2" class="tab">
    <!-- Содержимое вкладки 2 -->
  </div>
  <div id="tab_content3" class="tab">
    <!-- Содержимое вкладки 3 -->
  </div>
  <div id="tab_content4" class="tab">
    <!-- Содержимое вкладки 4 -->
  </div>
</div>
```

Представление блокнота

Все стили, которые зададут внешний вид блокнота, мы поместим в таблицу стилей pc.css. Поскольку блокнот нужен нам лишь на страницах создания и правки статьи, а в «мобильной» редакции сайта мы принципиально не станем давать возможность работать с внутренними данными сайта, то нет нужды делать блокнот доступным в обеих его редакциях.

Стилей для представления блокнота нам потребуется создать довольно много. Так что рассмотрим их по очереди.

- Для блока с корешками вкладок указываем внешний отступ сверху, чтобы визуально отделить его от предыдущего элемента.

  Поскольку гиперссылки, которые будут помещаться в этом блоке и создадут корешки вкладок, будут преобразованы нами в плавающие блочные элементы, сдвинутые к левому краю родителя, и поскольку мы не задали для блока корешков высоту явно, обязательно случится так, что гиперссылки по вертикали вылезут за пределы этого блока. Растянуть блок по высоте так, чтобы он вместил все корешки, поможет задание значения `hidden` для атрибута стиля `overflow` (этот прием мы уже использовали ранее).

  Еще мы задаем для блока корешков относительное позиционирование и достаточно большое значение z-индекса, чтобы расположить его выше других позиционируемых элементов (зачем это нужно, будет рассказано далее):

```
.tab-header { margin-top: 20px;
              overflow: hidden;
              position: relative;
              z-index: 99; }
```

- Для самих корешков мы указываем поведение как у блочных элементов, сдвиг к левой стороне родителя, внутренние отступы, внешний отступ справа, чтобы отделить корешки друг от друга, и рамку:

```
.tab-header a { display: block;
                float: left;
                padding: 5px 8px;
                margin-right: 3px;
                border: 1px solid grey; }
```

- Убираем у корешков подчеркивание текста:

```
.tab-header a:link, .tab-header a:visited, .tab-header a:active,
.tab-header a:hover {
  text-decoration: none;
}
```

- Задаем для корешков цвета текста и фона. В обычном состоянии белый текст будет выводиться на сером фоне, а при наведении курсора мыши — черный текст на оранжевом фоне:

```
.tab-header a:link, .tab-header a:visited {
  color: white;
  background-color: grey;
  transition: color 1s, background-color 1s;
}
.tab-header a:active, .tab-header a:hover {
  color: black;
  background-color: orange;
  transition: color 0.2s, background-color 0.2s;
}
```

- Для активного корешка (с привязанным стилевым классом `active`) задаем белый текст, прозрачный фон и белую рамку из одной нижней стороны — чтобы такой корешок визуально составлял единое целое с активной рамкой.

  К сожалению, если мы зададим для рамки корешка прозрачный цвет, сквозь нее будет просвечивать рамка, указанная для блока вкладок, и желаемого эффекта мы не достигнем. Но заданная нами тонкая белая рамка будет практически не заметна на светлом фоне, и в результате посетителю будет казаться, что активные корешок и рамка составляют единое целое:

```
.tab-header a.active:link, .tab-header a.active:visited,
.tab-header a.active:active, .tab-header a.active:hover {
  color: black;
  background-color: transparent;
  border-bottom-color: white;
}
```

- Блок вкладок получит рамку со всех сторон и расположение ниже всех плавающих элементов. Также мы указали для него относительное позиционирование и сдвиг вверх относительно своего изначального местоположения на значение толщины рамки — на один пиксел. Это нужно для того, чтобы убрать любые просветы между блоками корешков и вкладок:

```
.tab-content { border: 1px solid grey;
               clear: both;
               position: relative;
               top: -1px; }
```

- Блок отдельной вкладки мы растягиваем по горизонтали на все свободное пространство родителя минус величины внутренних отступов. Еще мы указываем для него абсолютное позиционирование, чтобы вкладки накладывались друг на друга, полную прозрачность и невидимость:

```
.tab-content div.tab { width: calc(100% - 20px);
                       padding: 10px;
                       position: absolute;
                       opacity: 0;
                       visibility: hidden;
                       transition: opacity 0.5s; }
```

- А активная вкладка будет полностью непрозрачной и видимой:

```
.tab-content div.tab-visible { opacity: 1;
                               visibility: visible;
                               transition: opacity 0.5s; }
```

Программирование блокнота

Весь код сценариев, которые зададут поведение для блокнота, мы поместим в отдельный файл, который назовем pc.js. Привяжем его к страницам редакции сайта, предназначенной для традиционных компьютеров, при помощи тега:

```
<script src="pc.js"></script>
```

И не забудем также поместить весь написанный в нем код в функцию-обработчик события `load` окна:

```
window.addEventListener("load", function() {
  // Сценарии
});
```

Чтобы заставить блокнот работать, нам понадобятся три сценария, два из которых будут выполнять инициализацию.

Первый из сценариев выполнит инициализацию блоков вкладок (листинг 37.8).

Листинг 37.8

```
var arrTabContents = document.querySelectorAll↵
("section div.tab-content");
var el, a, arr, n;
```

```
for (var i = 0; i < arrTabContents.length; i++) {
  el = arrTabContents[i];
  arr = el.querySelectorAll("div.tab");
  n = 0;
  for (var j = 0; j < arr.length; j++) {
    a = arr[j];
    if (a.clientHeight > n) n = a.clientHeight;
  }
  el.style.height = n + "px";
}
```

Сначала мы получаем доступ ко всем блокам с вкладками. (На тот случай, если мы захотим создать на странице несколько блокнотов, напишем универсальный код.) В каждом таком блоке мы перебираем отдельные блоки-вкладки и определяем высоту самого высокого из них. Это значение мы зададим в качестве высоты самого блока с вкладками, чтобы при переключении между вкладками высота самого блокнота не менялась.

Второй, совсем небольшой, сценарий выполнит инициализацию блоков корешков:

```
var arrTabs = document.querySelectorAll("section div.tab-header a");
for (var i = 0; i < arrTabs.length; i++)
  arrTabs[i].addEventListener("click", tabHeaderClick);
```

Здесь мы получаем доступ ко всем корешкам и привязываем к каждому обработчик события щелчка мышью — функцию `tabHeaderClick`, код которой приведен в листинге 37.8.

Листинг 37.8

```
function tabHeaderClick(evt) {
  var el = evt.target;
  var a = document.querySelector("section div.tab-visible");
  a.className = "tab";
  var s = el.id;
  var n = s.substr(0, s.indexOf("_"));
  var i = s.substr(s.length - 1, 1);
  a = document.getElementById(n + "_content" + i);
  a.className = "tab tab-visible";
  a = el.parentElement.querySelector("a.active");
  a.className = "";
  el.className = "active";
}
```

Сначала мы ищем активную вкладку, т. е. с дополнительно привязанным стилевым классом `tab-visible`, и привязываем к ней лишь стилевой класс `tab`, делая ее неактивной. Далее мы извлекаем из имени корешка, на котором был выполнен щелчок мышью, имя блокнота и номер его вкладки, соответствующей этому корешку, на-

ходим нужную вкладку и привязываем к ней стилевые классы `tab` и `tab-visible`. Наконец, мы убираем у активного ранее корешка привязку стилевого класса `active` и привязываем его к корешку, на котором посетитель щелкнул мышью.

Метод `indexOf` объекта `String` ищет в строке, у которой был вызван, символ, переданный ему единственным аргументом, и возвращает номер, под которым переданный символ присутствует в этой строке. Если такой символ не был найден, возвращается значение –1. А метод `substr` того же объекта был описан в *главе 16*.

Созданный нами блокнот можно увидеть на рис. 37.4. На активной в данный момент вкладке автор книги поместил Web-форму в качестве наглядного примера того, что вкладка созданного нами блокнота может иметь любое содержимое.

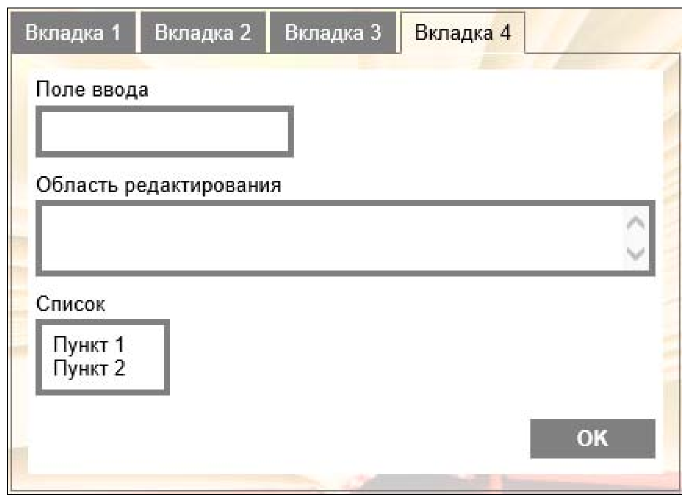
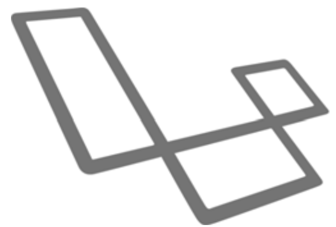


Рис. 37.4. Блокнот

ГЛАВА 38



Статические Web-страницы

Покончив с дизайном страниц и интерактивными элементами, можно приступать к программированию серверной части сайта.

Обычно в первую очередь создают серверные программы, выводящие статические страницы, — те, что содержат информацию, которая не берется из базы данных. К таким страницам относятся, прежде всего, главная страница сайта, страницы сведений о разработчиках сайта и о политике публикации статей, всевозможные справочные страницы и проч.

Маршруты

Начнем с создания необходимых маршрутов. Откроем файл routes\web.php и удалим весь имеющийся там программный код (все равно изначально присутствующие там маршруты нам не нужны). После чего добавим три следующих выражения:

```
Route::get('/', 'MainController@index');
Route::get('/policy', 'MainController@policy');
Route::get('/about', 'MainController@about');
```

Главная страница сайта будет выводиться действием `index` контроллера `MainController`. Страницы политики публикации и сведений о сайте будут иметь интернет-адреса **/policy** и **/about** и выводиться действиями `policy` и `about` того же контроллера.

Контроллеры

Мы делаем две редакции сайта: предназначенную для традиционных компьютеров и рассчитанную на мобильные устройства. Следовательно, нам понадобятся два родительских шаблона, на основе которых будут генерироваться страницы той или иной редакции сайта.

Базовый класс контроллера. Определение обращения с мобильного устройства

При этом нам потребуется как-то определять, откуда посетитель зашел на сайт: с традиционного компьютера или мобильного устройства. Это необходимо для того, чтобы выбрать нужный шаблон.

К счастью, в составе любого запроса всегда присутствуют сведения об отправившем его Web-обозревателе. Эти сведения доступны в качестве значения параметра `User-Agent`, а извлечь их поможет описанный в *главе 30* метод `header` класса `Request`.

В составе сведения о Web-обозревателе присутствует строка, обозначающая устройство, на котором запущен этот Web-обозреватель. Так, для Apple iPhone это будет строка `"iphone"`, для Apple iPod — строка `"ipod"`, а для устройства, работающего под управлением Google Android, — строка `"android"`.

Соответственно, нам достаточно получить сведения о Web-обозревателе и просмотреть их на предмет наличия одной из строк, идентифицирующих мобильное устройство. Для этого мы применим регулярное выражение, благо PHP, как мы знаем из *главы 24*, их поддерживает.

Удобнее всего поместить код, выясняющий род устройства, с которого был отправлен запрос, в класс `Controller` — родитель всех контроллеров. В таком случае все классы контроллеров получат нужную функциональность. Класс этот хранится в файле app\Http\Controllers\Controller.php и изначально не содержит никаких свойств и методов.

Создадим в классе `Controller` два свойства: свойство `isMobile` будет хранить логическую величину — признак того, было ли выполнено обращение с мобильного устройства (`true`) или с традиционного компьютера (`false`), а свойство `parentView` будет хранить строку с именем соответствующего родительского шаблона. Это позволит нам не выполнять проверку сведений о Web-обозревателе каждый раз, когда понадобится выяснить род устройства.

Исправим объявление класса `Controller`, как показано в листинге 38.1.

Листинг 38.1

```
class Controller extends BaseController {
  use AuthorizesRequests, DispatchesJobs, ValidatesRequests;

  public $isMobile = false;
  public $parentView;

  public function __construct() {
    $userAgent = strtolower(request()->header("User-Agent"));
    if (preg_match("/phone|iphone|itouch|ipod|symbian|android|htc_|" .
    "htc-|palmos|blackberry|opera mini|iemobile|windows ce|nokia|" .
```

```
    "fennec|hiptop|kindle|mot |mot-|webos\/|samsung|sonyericsson|" .
    "^sie-|nintendo/", $userAgent)) {
      $this->isMobile = true;
    } else {
      if (preg_match("/mobile|pda;|avantgo|eudoraweb|minimo|netfront|" .
      "brew|teleca|lg;|lge |wap;| wap /", $userAgent)) {
        $this->isMobile = true;
      }
    }
    if ($this->isMobile) {
      $this->parentView = "layouts.mobile";
    } else {
      $this->parentView = "layouts.pc";
    }
  }
}
```

Здесь нас интересует только код конструктора класса, в котором выполняется обработка сведений о Web-обозревателе и заполняются оба объявленных нами свойства.

Сначала мы приведем полученные сведения к нижнему регистру, воспользовавшись функцией PHP `strtolower`, принимающей в качестве единственного аргумента строку, которую следует преобразовать. Далее мы выясняем, совпадают ли эти сведения с двумя регулярными выражениями.

- Первое регулярное выражение включает строки, идентифицирующие наиболее распространенные мобильные устройства. Если это регулярное выражение совпадает со сведениями о Web-обозревателе, мы присваиваем свойству `isMobile` значение `true`.
- Если же первое регулярное выражение не совпадает со сведениями, мы сравниваем их со вторым регулярным выражением, которое содержит строки, идентифицирующие малораспространенные устройства, — в случае совпадения мы также присваиваем свойству `isMobile` значение `true`.
- Если оба сравнения не увенчались успехом, свойство `isMobile` получит значение `false`, заданное при его объявлении.

Для сравнения строки с регулярным выражением мы используем функцию PHP `preg_match`:

```
preg_match(<регулярное выражение>, <сравниваемая строка>)
```

Оба аргумента задаются в виде строк. Функция возвращает 1, если строка совпадает с регулярным выражением, и 0 в противном случае.

Напоследок мы задаем шаблон разметки layouts\mobile.blade.php, если обращение было выполнено с мобильного устройства, и layouts\pc.blade.php — если с традиционного компьютера. (Пути записаны относительно папки resources\views.)

Контроллер *MainController*

Теперь займемся контроллером `MainController`. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду:

```
php artisan make:controller MainController
```

Найдем в папке app\Http\Controllers созданный в результате выполнения этой команды файл MainController.php и откроем его. Добавим в него код, объявляющий действия `index`, `policy` и `about`, как показано в листинге 38.2.

Листинг 38.2

```
class MainController extends Controller {
  public function index() {
    return view("main");
  }

  public function policy() {
    return view("policy");
  }

  public function about() {
    return view("about");
  }
}
```

Здесь все просто: все объявленные нами три действия лишь выводят на экран соответствующую страницу с применением определенного шаблона.

В папке app\Http\Controllers также присутствует файл HomeController.php, хранящий код шаблона `HomeController`. Этот файл формируется при создании нового проекта. Нам он не нужен, и мы можем удалить его.

Шаблоны

Теперь начнем писать шаблоны. Только сначала выполним некоторые подготовительные действия:

- скопируем все созданные в *главах 36* и *37* внешние таблицы стилей (basic.css, pc.css, mobile.css и print.css) и фоновое изображение background.jpg в папку public\css;
- скопируем все созданные в тех же главах файлы сценариев (main.js и pc.js) в папку public\js.

Родительские шаблоны

Все шаблоны, на основе которых создаются страницы нашего сайта, будут потомками двух родительских шаблонов: это шаблон pc.blade.php, предназначенный

для «компьютерной» редакции сайта, и шаблон mobile.blade.php, предназначенный для его «мобильной» редакции.

Заглянем в папку resources\views\layouts — как мы помним из *главы 31*, именно там хранятся родительские шаблоны. Удалим оттуда все уже присутствующие там файлы шаблонов.

Код родительского шаблона pc.blade.php приведен в листинге 38.3 — как видим, он базируется на коде созданной нами в *главе 36* страницы pc.html.

Листинг 38.3

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="/css/basic.css" type="text/css">
    <link rel="stylesheet" href="/css/pc.css" type="text/css">
    <link rel="stylesheet" href="/css/print.css" type="text/css"
    media="print">
    <script src="/js/main.js"></script>
    @stack("head")
    <title>@yield("title") :: СЭП</title>
  </head>
  <body>
    <div id="container"></div>
    <header>
      <div id="header_1"><h1>СЭП</h1></div>
      <div id="header_2"><h2>сайт</h2></div>
      <div id="header_3"><h2>электронных</h2></div>
      <div id="header_4"><h2>публикаций</h2></div>
    </header>
    <nav>
      <?php $p = request()->path(); ?>
      <a href="/" @if ($p == "/") class="active" @endif>Главная</a>
      <a href="/policy" @if ($p == "policy") class="active"
      @endif>Политика</a>
      <a href="/about" @if ($p == "about") class="active" @endif>О
      сайте</a>
    </nav>
    <section>
      @if (session('status'))
        <p>{{ session('status') }}</p>
      @endif
      @yield("main")
    </section>
    <footer>
      <p>© коллектив разработчиков сайта, 2017 г.</p>
```

```
    </footer>
  </body>
</html>
```

Здесь мы создали секции `title`, в которой будет выводиться основная часть названия страницы, и `main`, где выводится основное содержание. Также мы создали стек `head`, куда мы поместим тег, привязывающий файл сценария pc.js (этот файл хранит код, применяемый только на страницах «компьютерной» редакции сайта).

В элементе основного содержания мы будем выводить всплывающее сообщение с именем `status`. Мы поместили выполняющий эту задачу код в родительский шаблон, так как сообщения подобного рода будут выводиться на многих, если не на всех, страницах.

Еще обратим внимание на код, создающий гиперссылки панели навигации. С помощью метода `path` класса `Request` мы получаем путь к серверной программе, входящий в состав интернет-адреса (этот метод был описан в *главе 30*). Далее мы сравниваем этот путь и интернет-адрес очередной создаваемой гиперссылки и, в случае их совпадения, привязываем к гиперссылке стилевой класс `active`, указывая таким образом, что таковая указывает на текущую страницу.

Код родительского шаблона mobile.php можно увидеть в листинге 38.4.

Листинг 38.4

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="/css/basic.css" type="text/css">
    <link rel="stylesheet" href="/css/mobile.css" type="text/css">
    <link rel="stylesheet" href="/css/print.css" type="text/css"
    media="print">
    <script src="/js/main.js"></script>
    <title>@yield("title") :: СЭП</title>
  </head>
  <body>
    <header>
      <h1>СЭП: сайт электронных публикаций</h1>
    </header>
    <section>
      @yield("main")
    </section>
    <footer>
      <p>© коллектив разработчиков сайта, 2017 г.</p>
    </footer>
  </body>
</html>
```

Здесь практически то же самое, что и в аналогичном шаблоне для «компьютерной» редакции сайта, за исключением отсутствия панели навигации и стека `head` (сценарии, хранящиеся в файле pc.js, на страницах «мобильной» редакции сайта нам не понадобятся никогда).

Шаблоны страниц

Шаблоны статических страниц мы будем хранить непосредственно в папке resources\views. Только предварительно очистим ее от всех уже присутствующих в ней файлов.

Шаблон главной страницы сайта получит имя main.blade.php. Его код представлен в листинге 38.5.

Листинг 38.5

```
@extends(request()->route()->getController()->parentView)

@section("title", "Главная")
@section("main")
  <h1>СЭП: сайт электронных публикаций</h1>
  <p>На этом сайте вы можете найти статьи по различным темам.</p>
  <p>Авторы статей могут опубликовать свои материалы. По всем вопросам,
  связанным с предоставлением доступа для публикации статей, просьба
  обращаться к <a href="mailto:admin@site.ru">администратору
  сайта</a>.</p>
  @if (request()->route()->getController()->isMobile)
    <ul>
      <li><a href="/policy">Политика публикации статей</a></li>
      <li><a href="/about">Сведения о сайте</a></li>
    </ul>
  @endif
@endsection
```

Здесь мы выводим гиперссылку, указывающую на адрес электронной почты администратора. (Изначально мы указали адрес, что называется, взятый с потолка, но перед публикацией сайта мы, разумеется, исправим его на реальный.) А если посетитель зашел на сайт с мобильного устройства, мы также выведем гиперссылки на страницы со сведениями о политике публикации и о самом сайте, — поскольку на страницах «мобильной» редакции нет панели навигации, эти гиперссылки будут весьма кстати.

Код шаблона страницы со сведениями о политике публикации статей policy.blade.php показан в листинге 38.6.

Листинг 38.6

```
@extends(request()->route()->getController()->parentView)

@section("title", "Политика публикации")
@section("main")
  @if (request()->route()->getController()->isMobile)
    <p><a href="/">Главная</a></p>
  @endif
  <h1>Политика публикации</h1>
  <p>Все статьи, опубликованные на этом сайте, являются собственностью
  их авторов.</p>
@endsection
```

Если обращение к сайту выполняется с мобильного устройства, мы дополнительно выводим гиперссылку, ведущую на главную страницу, чтобы посетитель смог вернуться в начало сайта.

Шаблон страницы со сведениями о самом сайте about.blade.php можно сделать на основе шаблона policy.blade.php (см. листинг 38.6). Они будут различаться лишь текстом.

Тестирование «мобильной» редакции Web-сайта

Если редакцию сайта, предназначенную для традиционных компьютеров, мы можем без труда протестировать в любом Web-обозревателе, то проверить работу «мобильной» редакции на обычном компьютере довольно проблематично. Нам понадобятся особые программные инструменты.

Автор рекомендует применять для тестирования «мобильной» редакции сайта Web-обозреватель Mozilla Firefox с установленным дополнительным модулем User-Agent Switcher. Этот модуль можно найти в магазине приложений Mozilla, загрузить и установить непосредственно в среде Firefox.

> ***Внимание!***
>
> Не путайте это дополнение User-Agent Switcher с аналогичным по назначению, но сильно устаревшим дополнением User Agent Switcher (в ее названии отсутствует символ дефиса).

После установки дополнительного модуля User-Agent Switcher в панели инструментов Firefox появится новая кнопка. Если нажать эту кнопку, откроется всплывающее меню (рис. 38.1), в котором можно выбрать тип эмулируемого устройства: смартфон на операционной системе Apple iOS, смартфон на Google Android и др.

Чтобы вернуться к обычному режиму просмотра сайтов, следует выбрать в упомянутом ранее меню пункт **Switch to Default User-Agent** (она расположено в левом нижнем углу) или нажать комбинацию клавиш <Ctrl>+<Shift>+<D>.

На рис. 38.2 показана главная страница «мобильной» редакции нашего сайта, отображаемая в Firefox в режиме эмуляции смартфона на Google Android.

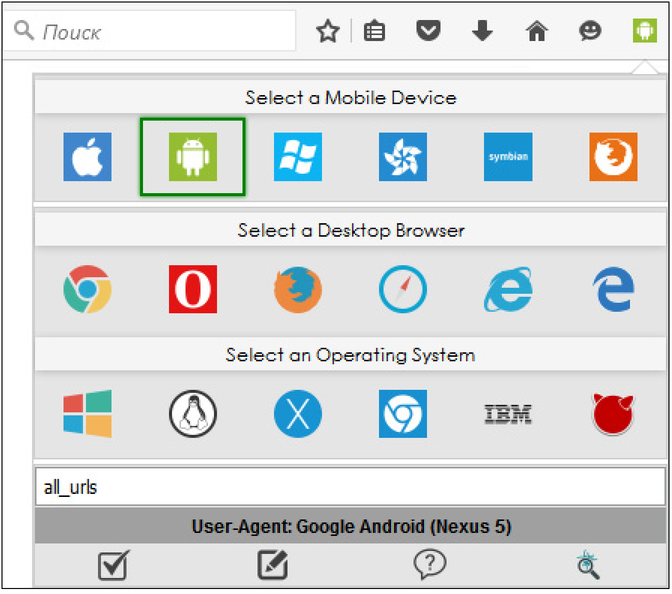


Рис. 38.1. Кнопка User-Agent Switcher и всплывающее меню, появляющееся при нажатии на нее

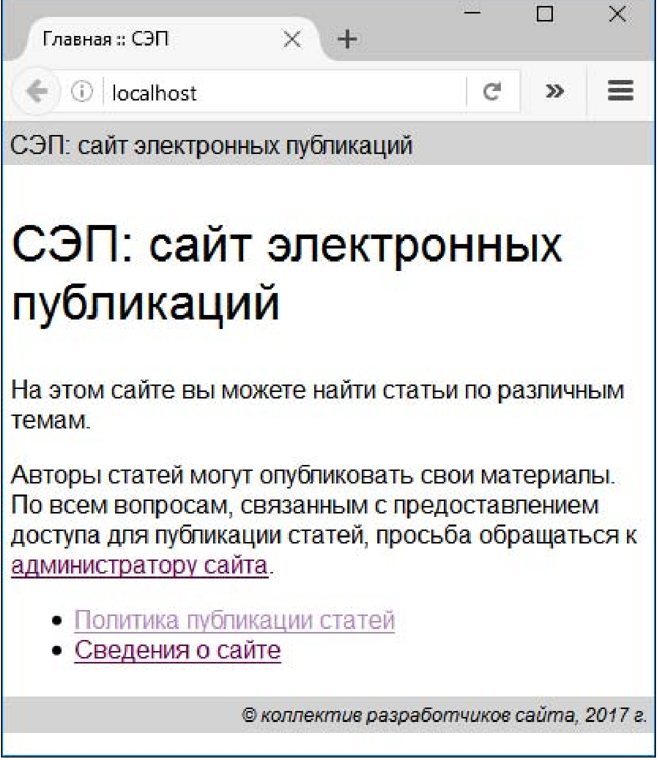
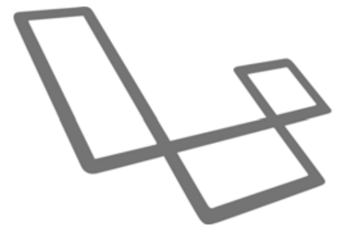


Рис. 38.2. Главная страница «мобильной» редакции сайта, отображаемая в режиме эмуляции мобильного устройства



ГЛАВА 39

Разграничение доступа и список пользователей

Перед тем как начать создавать страницы, на которых выводятся взятые из базы данные, необходимо позаботиться о реализации разграничения доступа. Мы уже знаем, что далеко не все внутренние данные сайта следует делать доступными для всех, и не все внутренние данные имеет смысл делать доступными даже для всех зарегистрированных на сайте пользователей.

К счастью, Laravel уже поставляется с мощной подсистемой разграничения доступа в комплекте, а каждый создаваемый проект включает часть ее реализации. Так что нам остается лишь немного исправить имеющееся под свои нужды.

Помимо этого, мы предусмотрим вывод списка всех зарегистрированных на сайте пользователей, предоставим возможность изменить сведения о пользователе и удалить его. Последнее позволит администратору сайта удалить пользователя, нарушившего правила сайта.

Маршруты

Откроем файл routes\web.php и добавим в него следующие выражения:

```
Auth::routes();
Route::get('/logout', 'Auth\LoginController@logout');
Route::get('/users', 'UserController@index');
Route::get("/users/{user}/edit", "UserController@input");
Route::put("/users", "UserController@save");
Route::get("/users/{user}/delete", "UserController@destroy");
```

Первые два выражения задают маршруты для действий, выполняющих вход, выход, регистрацию нового пользователя и сброс пароля. Они уже знакомы нам по *главе 33*.

Следующие выражения указывают на действия еще не созданного контроллера `UserController`:

- `index` — выводящее список пользователей;
- `input` — выводящее страницу для изменения данных о пользователе;

- `save` — выполняющее сохранение пользователя;
- `destroy` — выполняющее удаление пользователя.

Чтобы упростить код действий, которые будут сохранять и удалять пользователей, мы использовали в своих маршрутах внедрение модели (подробности — в *главе 29*).

Миграция и модель

Каждый вновь созданный проект содержит в папке database\migrations две миграции, генерирующие, соответственно, таблицу списка пользователей и таблицу для хранения идентификаторов запросов на сброс данных. Первая миграция хранится в файле *<дата и время>*_create_users_table.php, вторая — в файле *<дата и время>*_create_password_resets_table.php.

В таблице списка пользователей нам понадобится хранить роль пользователя, задающую его полномочия. Для этого мы добавим в эту таблицу поле `role` типа `STRING` длиной в один символ.

Поле `role` будет хранить одно из следующих значений:

- `"a"` — автор, имеет право на добавление статей, правку и удаление статей, написанных им самим;
- `"e"` — редактор, помимо этого, имеет право править и удалять статьи любых авторов, править и удалять комментарии;
- `"m"` — администратор, помимо этого, имеет право добавлять, править и удалять категории, подкатегории и пользователей.

Символ `"a"` мы укажем для этого поля в качестве значения по умолчанию.

Чтобы сделать все это, мы откроем файл, в котором хранится миграция, создающая список пользователей, и исправим код метода `up` класса `CreateUsersTable`, как показано в листинге 39.1 (добавленное выражение выделено полужирным шрифтом).

Листинг 39.1

```
class CreateUsersTable extends Migration {
  public function up() {
    Schema::create('users', function (Blueprint $table) {
      . . .
      $table->string('password');
      $table->string('role', 1)->default('a');
      $table->rememberToken();
      . . .
    });
  }
  . . .
}
```

После чего выполним обе миграции, открыв консоль OpenServer и отдав команду:

```
php artisan migrate
```

Осталось исправить код класса модели `User`, которая также создается при генерировании нового проекта. Нам нужно добавить поле `role` в список полей, вовлеченных в массовое присваивание (за подробностями — к *главе 28*). Откроем файл app\User.php и исправим объявление свойства `fillable` класса `User`, чтобы оно выглядело так:

```
protected $fillable = ['name', 'email', 'password', 'role'];
```

Служебные страницы

Теперь займемся страницами, выполняющими служебные задачи: вход на сайт, регистрацию нового пользователя и сброс пароля.

Регистрация нового пользователя

Первым делом исправим шаблон страницы регистрации, хранящийся в файле resources\views\auth\register.blade.php. Исправленный код этого шаблона показан в листинге 39.2.

Листинг 39.2

```
@extends("layouts.pc")

@section("title", "Регистрация")
@section("main")
  <h1>Регистрация нового пользователя</h1>
  <form method="POST" action="/register">
    {{ csrf_field() }}
    <label>Имя</label>
    <input type="text" name="name" value="{{ old('name') }}" required
    autofocus>
    @include("common.errors", ["el" => "name"])
    <label>E-mail</label>
    <input type="email" name="email" value="{{ old('email') }}" required>
    @include("common.errors", ["el" => "email"])
    <label>Пароль</label>
    <input type="password" name="password" required>
    @include("common.errors", ["el" => "password"])
    <label>Подтверждение пароля</label>
    <input type="password" name="password_confirmation" required>
    <label>Введите символы с изображения</label>
    <div><img src="{{ captcha_src() }}"></div>
    <input type="text" name="captcha" required>
```

```
    @include("common.errors", ["el" => "captcha"])
    <input type="submit" value="Зарегистрировать">
  </form>
@endsection
```

Мы добавили в форму регистрации изображение CAPTCHA и поле для ввода символов с него. Таким образом мы защитим сайт от спамеров.

Для вывода сообщений об ошибках мы используем вложенный шаблон common\errors.blade.php. Ему мы будем дополнительно передавать в контексте данных значение `el` — им станет имя элемента управления, для которого требуется вывести сообщение об ошибке.

Создадим в папке resources\views папку common, поместим в нее файл errors.blade.php со следующим кодом:

```
@if ($errors->has($el))
  <div class="errors">{{ $errors->first($el) }}</div>
@endif
```

Зададим представление для блока, где будут выводиться сообщения об ошибках. Добавим в таблицу стилей basic.css такой стиль:

```
form div.errors { color: red; }
```

Он укажет красный цвет для текста сообщений об ошибках.

Теперь исправим интернет-адрес, по которому будет выполняться перенаправление после успешной регистрации пользователя. Откроем файл app\Http\Controllers\Auth\RegisterController.php и исправим объявление свойства `redirectTo` класса `RegisterController`:

```
protected $redirectTo = '/';
```

Напоследок укажем свой собственный текст сообщений об ошибках. Найдем в объявлении того же класса `RegisterController` защищенный метод `validator` и исправим его код, чтобы он выглядел, как представлено в листинге 39.3.

Листинг 39.3

```
protected function validator(array $data) {
  return Validator::make($data, [
    'name' => 'required|max:255',
    'email' => 'required|email|max:255|unique:users',
    'password' => 'required|min:6|confirmed',
    'captcha' => 'captcha',
  ], [
    'name.required' => 'Введите имя пользователя',
    'name.max' => 'Имя пользователя должно быть не длиннее 255 символов',
    'email.required' => 'Введите адрес электронной почты',
    'email.email' => 'Введите корректный адрес электронной почты',
```

```
    'email.max' => 'Адрес электронной почты не должен превышать в ' .
    'длину 255 символов',
    'email.unique' => 'Пользователь с таким адресом электронной почты ' .
    'уже зарегистрирован в списке',
    'password.required' => 'Введите пароль',
    'password.min' => 'Пароль должен включать не менее 6 символов',
    'password.confirmed' => 'В полях "Пароль" и "Подтверждение ' .
    'пароля"  следует указать одно и то же значение',
    'captcha.captcha' => 'Введены не те символы',
  ]);
}
```

Вход на Web-сайт

Опять же, начнем с правки шаблона страницы входа, что хранится в файле resources\views\auth\login.blade.php. Исправленный код этого шаблона показывает листинг 39.4.

Листинг 39.4

```
@extends("layouts.pc")

@section("title", "Вход")
@section("main")
  <h1>Вход на сайт</h1>
  <form method="POST" action="/login">
    {{ csrf_field() }}
    <label>E-mail</label>
    <input type="email" name="email" value="{{ old('email') }}" required
    autofocus>
    @include("common.errors", ["el" => "email"])
    <label>Пароль</label>
    <input type="password" name="password" required>
    @include("common.errors", ["el" => "password"])
    <label>Запомнить меня</label>
    <input type="checkbox" name="remember"
    {{ old('remember') ? 'checked' : ''}}>
    <input type="submit" value="Войти">
  </form>
  <p><a href="/password/reset">Забыли пароль?</a></p>
@endsection
```

Исправим интернет-адрес, по которому будет выполняться перенаправление после успешного входа на сайт. Для этого откроем файл app\Http\Controllers\Auth\LoginController.php и изменим объявление свойства `redirectTo` класса `LoginController`:

```
protected $redirectTo = '/';
```

И внесем в код класса `LoginController` дополнения, задающие наши собственные сообщения об ошибках и описанные в *разд. «Базовые настройки» главы 33*. Код, который следует добавить, приведен в листинге 33.5.

Процедура сброса пароля

Здесь нам потребуется внести довольно много правок в несколько различных файлов и, помимо этого, создать дополнительные файлы.

Отправка письма со сведениями о сбросе пароля

Сначала исправим код шаблона email.blade.php, находящегося в папке resources\views\auth\passwords, — он формирует страницу для занесения адреса электронной почты, по которому будет отправлено письмо с гиперссылкой, ведущей на страницу сброса пароля. Новый код этого шаблона показан в листинге 39.5.

Листинг 39.5

```
@extends("layouts.pc")

@section("title", "Сброс пароля")
@section("main")
  <h1>Сброс пароля</h1>
  <form method="POST" action="/password/email">
    {{ csrf_field() }}
    <label>E-mail</label>
    <input type="email" name="email" value="{{ old('email') }}" required>
    @include("common.errors", ["el" => "email"])
    <input type="submit" value="Отправить письмо">
  </form>
@endsection
```

Откроем файл app\Http\Controllers\Auth\ForgotPasswordController.php, в котором хранится объявление класса `ForgotPasswordController`, и дополним его, указав собственные сообщения об ошибках. Все необходимые действия были описаны в *разд. «Базовые настройки» главы 33*, а добавляемый код приведен в листинге 33.6.

Собственно сброс пароля

Исправим шаблон resources\views\auth\passwords\reset.blade.php, который формирует страницу сброса пароля. После правки у нас должен получиться код, приведенный в листинге 39.6.

Листинг 39.6

```
@extends("layouts.pc")

@section("title", "Сброс пароля")
@section("main")
```

```
  <h1>Сброс пароля</h1>
  <form method="POST" action="/password/reset">
    {{ csrf_field() }}
    <input type="hidden" name="token" value="{{ $token }}">
    <label>E-mail</label>
    <input type="email" name="email" value="{{ $email or old('email') }}"
    required autofocus>
    @include("common.errors", ["el" => "email"])
    <label>Пароль</label>
    <input type="password" name="password" required>
    @include("common.errors", ["el" => "password"])
    <label>Подтверждение пароля</label>
    <input type="password" name="password_confirmation" required>
    <input type="submit" value="Сбросить пароль">
  </form>
@endsection
```

Укажем интернет-адрес для перенаправления после сброса пароля. Для этого в файле app\Http\Controllers\Auth\ResetPasswordController.php исправим объявление свойства `redirectTo` класса `ResetPasswordController`:

```
protected $redirectTo = '/';
```

Зададим свои сообщения об ошибках. Для этого добавим в объявление этого класса методы `sendResetResponse` и `sendResetFailedResponse`, чей код можно взять из листинга 33.7, и метод `validationErrorMessages`, код которого можно взять из листинга 39.7. (За подробностями обращайтесь к *разд. «Базовые настройки» главы 33*.)

Листинг 39.7

```
protected function validationErrorMessages() {
  return [
    'email.required' => 'Укажите адрес электронной почты',
    'email.email' => 'Укажите корректный адрес электронной почты',
    'password.required' => 'Введите пароль',
    'password.min' => 'Пароль должен включать не менее 6 символов',
    'password.confirmed' => 'В полях "Пароль" и "Подтверждение ' .
    'пароля" следует указать одно и то же значение',
  ];
}
```

Электронное письмо со сведениями для сброса пароля

Осталось создать оповещение, на основе которого будет генерироваться электронное письмо с гиперссылкой сброса пароля. Откроем консоль OpenServer, перейдем в папку проекта и отдадим команду:

```
php artisan make:notification PasswordReset
```

Исправим только что созданный файл app\Notifications\PasswordReset.php с кодом класса оповещения `PasswordReset`, чтобы он выглядел, как показано в листинге 39.8.

Листинг 39.8

```
<?php
namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class PasswordReset extends Notification {
  use Queueable;

  public $token;

  public function __construct($token) {
    $this->token = $token;
  }

  public function via($notifiable) {
    return ['mail'];
  }

  public function toMail($notifiable) {
    return (new MailMessage)
    ->greeting("Уважаемый пользователь!")
    ->line("Вы получили это письмо, поскольку произвели процедуру " .
    "сброса пароля.")
    ->action("Сбросить пароль", url("password/reset", $this->token))
    ->line("Если вы не выполняли сброс пароля, ничего не " .
    "предпринимайте.")
    ->from("admin@site.ru", "Администрация");
  }

  public function toArray($notifiable) {
    return [];
  }
}
```

В папке resources\views\vendor\notifications найдем два шаблона, формирующие электронное письмо, и также исправим их. Код измененного шаблона email-plain.blade.php (письмо в текстовом формате) показан в листинге 39.9, а код шаблона email.blade.php (письмо в формате HTML) — в листинге 39.10.

Листинг 39.9

```
<?php
if (! empty($greeting)) {
  echo $greeting, "\n\n";
}

if (! empty($introLines)) {
  echo implode("\n", $introLines), "\n\n";
}

if (isset($actionText)) {
  echo "{$actionText}: {$actionUrl}", "\n\n";
}

if (! empty($outroLines)) {
  echo implode("\n", $outroLines), "\n\n";
}

echo 'С наилучшими пожеланиями,', "\n";
echo 'администрация сайта.', "\n";
```

Листинг 39.10

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1><a href="{{ url('/') }}">Сайт электронных публикаций</a></h1>
    <h2>@if (! empty($greeting)) {{ $greeting }} @endif</h2>
    @foreach ($introLines as $line)
      <p>{{ $line }}</p>
    @endforeach
    @if (isset($actionText))
      <p style="font-size: larger;">
        <a href="{{ $actionUrl }}" target="_blank">{{ $actionText }}</a>
      </p>
    @endif
    @foreach ($outroLines as $line)
      <p>{{ $line }}</p>
    @endforeach
    <p>С уважением,<br>администрация сайта.</p>
  </body>
</html>
```

И, наконец, дополним код класса модели `User` (он находится в папке app):

```
. . .
use App\Notifications\PasswordReset;

class User extends Authenticatable {
  . . .
  public function sendPasswordResetNotification($token) {
    $this->notify(new PasswordReset($token));
  }
}
```

Инструменты для работы со списком пользователей

Создадим теперь набор административных страниц, предназначенных для работы с зарегистрированными пользователями: просмотра их списка, правки и удаления.

Выполнять все операции мы дадим право только пользователям с ролью администратора. Авторы и редакторы вообще не смогут получить доступ к этим инструментам.

Список пользователей

Работу со списком пользователей будет осуществлять отдельный контроллер `UserController`. Создадим его, воспользовавшись консолью OpenServer. Откроем ее, перейдем в папку проекта и скомандуем:

```
php artisan make:policy UserPolicy
```

Откроем созданный в результате выполнения этой команды файл app\Http\Controllers\UserController.php, в котором хранится объявление класса `UserController`, и добавим в него метод-действие `index`, которое выведет список пользователей на экран. Новый код этого класса можно увидеть в листинге 39.11.

Листинг 39.11

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

use Illuminate\Http\Request;
use App\User;

class UserController extends Controller {
  public function index() {
    $users = User::select("id", "email", "name", "role")
    ->orderBy("email")->get();
    return view("users.index", ["users" => $users]);
  }
}
```

Из таблицы списка пользователей мы извлекаем лишь номера, имена, адреса электронной почты и роли (поля `id`, `name`, `email` и `role`). Прочие поля мы извлекать не станем, поскольку они все равно не выводятся на экране, — так мы сэкономим оперативную память. Помимо этого, мы сортируем список пользователей по адресам электронной почты.

На странице нам понадобится выводить роль пользователя — является ли он автором, редактором или администратором — в читабельном виде. Можно написать соответствующий код прямо в шаблоне, но удобнее создать в классе модели `User` вычисляемое поле, из которого можно будет только извлекать значение. Откроем файл app\User.php и добавим в него метод `getFriendlyRoleAttribute`, создающий вычисляемое поле `friendly_role` (листинг 39.12).

Листинг 39.12

```
public function getFriendlyRoleAttribute() {
  switch ($this->attributes["role"]) {
    case "a":
      return "Автор";
      break;
    case "e":
      return "Редактор";
      break;
    case "m":
      return "Администратор";
      break;
  }
}
```

В папке resources\views создадим папку users и поместим в нее файл шаблона index.blade.php, который сформирует страницу списка пользователей. Содержимое этого шаблона показано в листинге 39.13.

Листинг 39.13

```
@extends("layouts.pc")

@section("title", "Пользователи")
@section("main")
  <h1>Список пользователей</h1>
  <table class="list">
    <tr>
      <th>E-mail</th>
      <th>Имя</th>
      <th>Роль</th>
      <th colspan="2"> </th>
    <tr>
```

```
    @foreach ($users as $user)
      <tr>
        <td>{{ $user->email }}</td>
        <td>{{ $user->name }}</td>
        <td class="center">{{ $user->friendly_role }}</td>
        <td class="links">
          <a href="{{ action('UserController@input',
          ['user' => $user->id]) }}">Исправить</a>
        </td>
        <td class="links">
          <a href="{{ action('UserController@destroy',
          ['user' => $user->id]) }}" class="adel">Удалить</a>
        </td>
      </tr>
    @endforeach
  </table>
@endsection
```

Список пользователей у нас будет выводиться в виде таблицы, включающей колонки с адресом электронной почты пользователя, его именем, наименованием его роли и гиперссылками на страницы правки и удаления. Для вывода наименования роли мы применяем только что созданное вычисляемое поле `friendly_name`. А, поскольку страница списка пользователей у нас будет доступна только в «компьютерной» редакции сайта, мы сразу укажем в качестве родительского шаблона layouts\pc.blade.php.

К гиперссылкам, выполняющим удаление пользователя, мы привяжем стилевой класс `adel`. Это нужно для того, чтобы впоследствии привязать к этим гиперссылкам обработчик события `click`, который будет предварительно спрашивать пользователя, действительно ли следует удалить эту запись.

Осталось добавить в таблицу стилей pc.css несколько стилей, которые зададут представление для списка пользователей. Эти стили приведены в листинге 39.14.

Листинг 39.14

```
table.list {                    width: 100%;
                                border-collapse: collapse; }
table.list th, table.list td { padding: 5px;
                                border: 1px solid grey; }
td.center, td.links {           text-align: center; }
td.links {                      width: 80px; }
```

Готовая страница списка пользователей показана на рис. 39.1. Поскольку панель навигации еще не включает соответствующей гиперссылки (мы доработаем ее позже), зайти на эту страницу можно лишь набрав интернет-адрес **http://localhost/users/**.

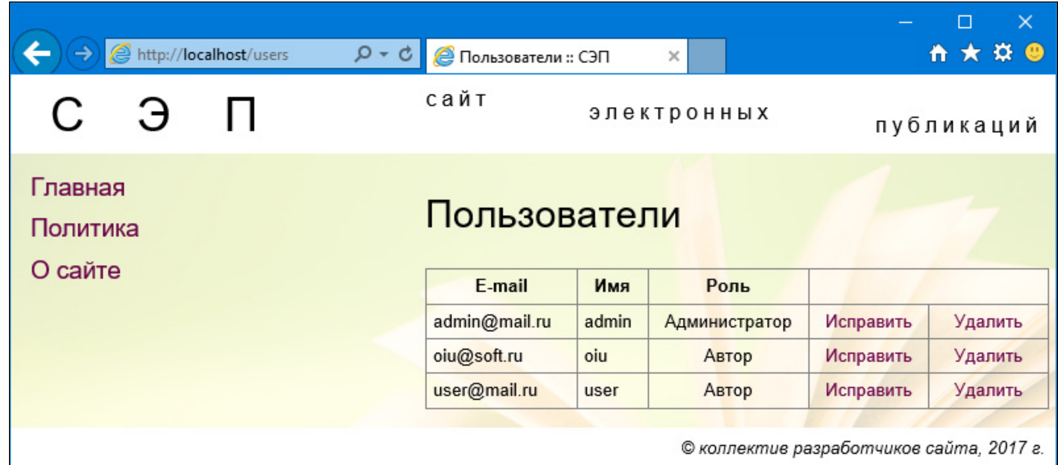


Рис. 39.1. Страница списка пользователей

Правка пользователя

Мы дадим администратору возможность изменить адрес электронной почты пользователя, его имя и, разумеется, роль. Пароль пользователь может изменить самостоятельно — с помощью средств сброса пароля.

Добавим в класс контроллера `UserController` методы-действия `input` и `save`. Их код можно увидеть в листинге 39.15. Также вставим в начало файла, где хранится объявление этого класса, выражение импортирования класса `App\Http\Requests\UserRequest` (этот класс запроса мы создадим очень скоро).

Листинг 39.15

```
. . .
use App\Http\Requests\UserRequest;

class UserController extends Controller {
  . . .
  public function input(User $user) {
    return view("users.input", ["user" => $user]);
  }

  public function save(UserRequest $request) {
    $user = User::findOrFail($request->id);
    $user->fill($request->all())->save();
    return redirect()->action("UserController@index")
    ->with("status", "Пользователь " . $user->name . " исправлен");
  }
}
```

Действие `input`, выводящее страницу правки пользователя, принимает в качестве параметра объект модели, представляющий выбранного пользователя. Здесь мы также применили внедрение модели.

Действие `save`, сохраняющее пользователя, дополнительно выводит всплывающее сообщение, уведомляющее об успешном сохранении пользователя.

Создадим класс запроса `UserRequest`, в котором опишем правила проверки занесенных данных на корректность. Откроем консоль OpenServer, перейдем в папку проекта и дадим команду:

```
php artisan make:request UserRequest
```

Откроем файл app\Http\Requests\UserRequest.php, где хранится код этого класса, и исправим его, как показано в листинге 39.16.

Листинг 39.16

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

use Illuminate\Foundation\Http\FormRequest;

class UserRequest extends FormRequest {
  public function authorize() {
    return true;
  }

  public function rules() {
    return [
      'name' => 'required|max:255',
      'email' => 'required|email|max:255',
    ];
  }

  public function messages() {
    return [
      'name.required' => 'Введите имя пользователя',
      'name.max' => 'Имя пользователя должно быть не длиннее ' .
      '255 символов',
      'email.required' => 'Введите адрес электронной почты',
      'email.email' => 'Введите корректный адрес электронной почты',
      'email.max' => 'Адрес электронной почты не должен превышать ' .
      'в длину 255 символов',
    ];
  }
}
```

И, наконец, напишем шаблон resources\views\users\input.blade.php, которой сформирует страницу правки пользователя (листинг 39.17).

Листинг 39.17

```
@extends("layouts.pc")

@section("title", $user->name . " :: Пользователи")
@section("main")
  <h1>Пользователь {{ $user->name }}</h1>
  <form method="POST" action="/users">
    {{ csrf_field() }}
    {{ method_field('PUT') }}
    <input type="hidden" name="id" value="{{ old("id", $user->id) }}">
    <label>Имя</label>
    <input type="text" name="name" value="{{ old('name', $user->name) }}"
    required>
    @include("common.errors", ["el" => "name"])
    <label>E-mail</label>
    <input type="email" name="email"
    value="{{ old('email', $user->email) }}" required>
    @include("common.errors", ["el" => "email"])
    <label>Роль</label>
    <?php $r = old('role', $user->role); ?>
    <select name="role">
      <option value="a" @if ($r == "a") selected @endif>Автор</option>
      <option value="e" @if ($r == "e") selected @endif>Редактор</option>
      <option value="m" @if ($r == "m") selected
      @endif>Администратор</option>
    </select>
    <input type="submit" value="Сохранить">
  </form>
  <p><a href="/users">Список пользователей</a></p>
@endsection
```

Здесь стоит обратить внимание на код, создающий пункты раскрывающегося списка ролей. В нем выполняется последовательное сравнение значения, взятого из модели, со строками `"a"`, `"e"` и `"m"`, и, если сравнение увенчается успехом, соответствующий пункт списка делается изначально выбранным.

Удаление пользователя

Удаление пользователя выполнит действие `delete` контроллера `User` (листинг 39.18).

Листинг 39.18

```
public function destroy(User $user) {
  $name = $user->name;
  $user->delete();
```

```
  return redirect()->action("UserController@index")
  ->with("status", "Пользователь " . $name . " удален");
}
```

Дополнительно обезопасимся от случайного удаления пользователя. Для этого при щелчке на гиперссылке **Удалить** на странице списка пользователей будем выводить окно-предупреждение. Пользователь будет удален только в том случае, если в этом окне будет нажата кнопка **OK**.

Откроем файл public\js\pc.js и добавим в тело обработчика события `load` окна код, представленный в листинге 39.19.

Листинг 39.19

```
function deleteLinkClick(evt) {
  if (!window.confirm("Удалить запись?"))
    evt.preventDefault();
}

var arrLinks = document.querySelectorAll("a.adel");
for (var i = 0; i < arrLinks.length; i++)
  arrLinks[i].addEventListener("click", deleteLinkClick);
```

Этот код ищет все гиперссылки с привязанным стилевым классом `adel` (это как раз и есть гиперссылки, указывающие на действие, которое выполняет удаление пользователя) и привязывает к ним обработчик события `click`. Этот обработчик выведет на экран упомянутое ранее окно-предупреждение и, если пользователь нажмет кнопку **Отмена** или просто закроет это окно, отменяет обработчик события по умолчанию — переход по гиперссылке.

Этот код является универсальным, и мы можем использовать его на других страницах — списках всевозможных позиций. Вообще, написание универсального кода — наилучший подход в Web-программировании.

В родительском шаблоне resources\views\layouts\pc.blade.php мы ранее создали стек `head`. Давайте откроем файл шаблона resources\views\users\index.blade.php, генерирующий страницу списка пользователей, и вставим в его начало, непосредственно после команды `@extends`, следующий фрагмент:

```
@push("head")
  <script src="/js/pc.js"></script>
@endpush
```

Он добавит в стек `head`, а значит, и в секцию заголовка этой страницы, тег, привязывающий файл сценариев public\js\pc.js.

Разграничение доступа

Закончив создавать страницы, предназначенные для работы с зарегистрированными пользователями, займемся ограничением доступа к ним. Мы предоставим такой доступ только администраторам. И используем для этого специально созданную политику `UserPolicy`.

Запустим консоль OpenServer, перейдем в папку проекта и отдадим команду:

```
php artisan make:policy UserPolicy
```

Откроем только что созданный файл app\Policies\UserPolicy.php, хранящий код этого класса, и исправим его таким образом, чтобы он выглядел, как показано в листинге 39.20.

Листинг 39.20

```
<?php
namespace App\Policies;

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

class UserPolicy {
  use HandlesAuthorization;

  public function manipulate(User $user) {
    return $user->role == "m";
  }
}
```

Здесь мы объявили метод `manipulate`, который задаст права доступа сразу для всех действий со списком пользователей: просмотру, правке и удалению. Как видим, он разрешит выполнять эти операции только администраторам (пользователям, имеющим роль `"m"`).

Откроем файл app\Providers\AuthServiceProvider.php, в котором хранится код класса провайдера `AuthServiceProvider`, и зарегистрируем в нем только что созданную нами политику, связав ее с моделью `User`. Для этого укажем в качестве единственного (пока) элемента ассоциативного массива, присваиваемого свойству `policies`, следующий элемент:

```
protected $policies = [
  'App\User' => 'App\Policies\UserPolicy',
];
```

После чего останется лишь открыть файл app\Http\Controllers\UserController.php (он хранит код класса контроллера `UserController`) и добавить в него конструктор:

```
public function __construct() {
  parent::__construct();
  $this->middleware("can:manipulate,App\User");
}
```

Обратим внимание, что в самом начале кода конструктора присутствует выражение вызова конструктора родительского класса, в нашем случае — `Controller`. Именно в нем мы записали код, определяющий род устройства, с которого был получен запрос (является ли оно традиционным компьютером или мобильным устройством). Если этого не сделать, в свойство `parentView` класса `Controller` не будет подставлено имя родительского шаблона, и при обращении к странице списка пользователей мы получим ошибку.

Кроме того, мы задали для всех действий этого контроллера посредник `can`, указав ему в качестве параметров операцию `manipulate` и полное имя класса модели `User`. Встретив это выражение, Laravel загрузит связанную с этой моделью политику `UserPolicy`, выполнит объявленный в ней метод `manipulate` и на основе возвращенного им результата определит, может ли текущий пользователь работать со списком пользователей или нет.

Панель навигации

Все страницы, контроллеры и действия, необходимые для реализации разграничения доступа и работы с зарегистрированными пользователями, готовы. Вот только попасть на них мы можем, лишь вручную набрав в Web-обозревателе их интернет-адреса…

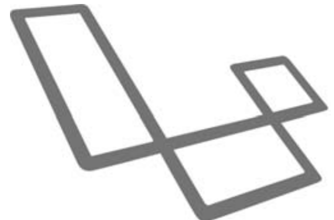
Откроем родительский шаблон resources\views\layouts\pc.blade.php и вставим в код, создающий панель навигации, фрагмент, показанный в листинге 39.21.

Листинг 39.21

```
<nav>
  . . .
  @can ("manipulate", "App\User")
    <a href="/users" @if ($p == "users") class="active"
    @endif>Пользователи</a>
  @endcan
  @if (auth()->check())
    <a href="/logout">Выход</a>
  @else
    <a href="/login" @if ($p == "login") class="active" @endif>Вход</a>
    <a href="/register" @if ($p == "register") class="active"
    @endif>Регистрация</a>
  @endif
</nav>
```

Отметим, что гиперссылка, указывающая на список пользователей, выводится только в том случае, если текущий пользователь имеет право выполнять операцию `manipulate`, заданную в политике `UserPolicy` (а такими правами у нас обладают лишь администраторы). Гиперссылка **Выход** выводится, только если пользователь выполнил вход на сайт, а гиперссылки **Вход** и **Регистрация** доступны лишь гостям.

Теперь протестируем сайт. Создадим с помощью страницы регистрации несколько пользователей. Запустим программу phpMyAdmin (о ней рассказано в *приложении 2*) и у одного из них укажем роль `"m"` (администратор). Выполним вход на сайт под именем этого пользователя, откроем список пользователей и попробуем исправить какую-либо из его позиций и выполнить удаление. После чего выйдем с сайта, войдем под именем другого пользователя, имеющего другую роль, и посмотрим, доступен ли нам список пользователей. Если нам удалось выполнить все эти действия, мы можем быть уверены, что подсистема разграничения доступа нашего сайта работает как надо.



ГЛАВА 40

Категории и подкатегории

На очереди — категории и подкатегории. Мы создадим страницы для управления ими, реализуем вывод их перечней на страницах «мобильной» редакции сайта и в панели навигации редакции «компьютерной». И дадим доступ к страницам управления категориями и подкатегориями только администраторам.

Маршруты

Откроем файл routes\web.php и добавим в него следующие выражения, задающие необходимые маршруты. Их много, и все они приведены в листинге 40.1.

Листинг 40.1

```
Route::get("/view/{category}", "CategoryController@view");
Route::get("/view/{category}/{subcategory}", "ArticleController@index");

Route::get("/categories", "CategoryController@index");
Route::get("/categories/create", "CategoryController@input")
->name("categories.create");
Route::post("/categories", "CategoryController@save");
Route::get("/categories/{category}/edit", "CategoryController@input");
Route::put("/categories", "CategoryController@save");
Route::get("/categories/{category}/delete",
"CategoryController@destroy");

Route::get("/subcategories", "SubcategoryController@index");
Route::get("/subcategories/create", "SubcategoryController@input")
->name("subcategories.create");
Route::post("/subcategories", "SubcategoryController@save");
Route::get("/subcategories/{subcategory}/edit",
"SubcategoryController@input");
Route::put("/subcategories", "SubcategoryController@save");
Route::get("/subcategories/{subcategory}/delete",
"SubcategoryController@destroy");
```

Первый маршрут связывает интернет-адрес вида **/view/<*категория*>** с действием `view` контроллера `CategoryController`, а второй — интернет-адрес вида **/view/<*категория*>/<*подкатегория*>** с действием `index` контроллера `ArticleController` (его мы создадим в *главе 41*). При этом в составе интернет-адресов будут передаваться не номера категорий и подкатегорий, а их слаги.

Слагом называется название какой-либо позиции (категории, подкатегории, статьи), записанное символами латиницы, цифрами, знаками подчеркивания и дефиса (дефисы обычно задействуются вместо пробелов). Слаги используются, чтобы сделать интернет-адреса читабельными для людей и более значимыми для поисковых интернет-служб.

Форматы этих двух интернет-адресов весьма примечательны — можно сказать, что они показывают иерархию категорий и подкатегорий. Подобные интернет-адреса в настоящее время наиболее востребованы.

Остальные маршруты аналогичны тем, что мы рассмотрели в *главе 32*, и особого интереса не представляют. Отметим, что в них также используется внедрение модели.

Миграции и модели

Для хранения списков категорий и подкатегорий мы создадим в базе данных сайта две таблицы, связанные между собой связью «один-ко-многим». Понятно, что таблица категорий станет первичной, а таблица подкатегорий — вторичной.

Миграция и модель списка категорий

Список категорий будет храниться в таблице `categories` базы данных. Ее структура представлена в табл. 40.1 (служебные поля, наподобие поля номера, времени создания и последнего изменения записи, в ней не показаны).

Таблица 40.1. *Структура таблицы* `categories`

Имя поля	Тип поля	Параметры поля	Назначение поля
`name`	`VARCHAR`	30 символов, обязательное, уникальный индекс	Название категории
`slug`	`VARCHAR`	30 символов, обязательное, уникальный индекс	Слаг категории
`order`	`TINYINT`	Беззнаковое, индекс, значение по умолчанию — 0	Порядок

Создадим миграцию для этой таблицы, запустив консоль OpenServer, перейдя в папку проекта и выполнив команду:

```
php artisan make:migration create_categories_table --create=categories
```

Найдем в папке database\migrations файл *<дата и время>*_create_categories_table.php, в котором хранится код класса миграции, откроем его и исправим код метода `up` этого класса (листинг 40.2).

Листинг 40.2

```
public function up() {
  Schema::create('categories', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name', 30)->unique();
    $table->string('slug', 30)->unique();
    $table->tinyInteger('order')->default(0)->unsigned()->index();
    $table->timestamps();
  });
}
```

Выполним миграцию, отдав в консоли OpenServer команду:

```
php artisan migrate
```

И сразу же создадим модель `Category`, скомандовав там же:

```
php artisan make:model Category
```

Откроем файл app\Category.php с классом вновь созданной модели и исправим код этого класса, как показано в листинге 40.3.

Листинг 40.3

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {
  protected $fillable = ["name", "slug", "order"];

  public function subcategories() {
    return $this->hasMany("App\Subcategory", "category", "id");
  }

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

Прежде всего, мы создали список полей, вовлеченных в массовое присваивание. После чего объявили метод, создающий связь с таблицей подкатегорий, предположив, что модель подкатегорий носит имя `Subcategory`, а участвующее в образова-

нии связи поле этой таблицы — имя `category`. И, наконец, указали поле для поиска записей при внедрении модели — `slug`, поскольку именно слаги, а не номера категорий будут присутствовать в составе интернет-адресов.

Миграция и модель списка подкатегорий

Займемся таблицей подкатегорий — `subcategories`. Ее структура показана в табл. 40.2 (опять же, служебные поля, наподобие поля номера, времени создания и последнего изменения записи, в ней опущены).

Таблица 40.2. *Структура таблицы* `subcategories`

Имя поля	Тип поля	Параметры поля	Назначение поля
name	VARCHAR	30 символов, обязательное, уникальный индекс	Название подкатегории
slug	VARCHAR	30 символов, обязательное, уникальный индекс	Слаг подкатегории
category	INT	Беззнаковое, индекс	Категория
order	TINYINT	Беззнаковое, индекс, значение по умолчанию — 0	Порядок

Создадим для таблицы миграцию:

```
php artisan make:migration create_subcategories_table ↳
--create=subcategories
```

Найдем в папке database\migrations файл *<дата и время>*_create_subcategories_table.php с объявлением класса миграции, откроем его и исправим код метода `up` (листинг 40.4).

Листинг 40.4

```
public function up() {
  Schema::create('subcategories', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name', 30)->unique();
    $table->string('slug', 30)->unique();
    $table->integer('category')->unsigned();
    $table->tinyInteger('order')->default(0)->unsigned()->index();
    $table->timestamps();
    $table->foreign('category')->references('id')->on('categories')
    ->onDelete('restrict')->onUpdate('restrict');
  });
}
```

Выполним миграцию:

```
php artisan migrate
```

Создадим модель `Subcategory`:

```
php artisan make:model Subcategory
```

Откроем файл app\Subcategory.php с классом модели и исправим его, как показано в листинге 40.5.

Листинг 40.5

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Subcategory extends Model {
  protected $fillable = ["name", "slug", "category", "order"];

  public function cat() {
    return $this->belongsTo("App\Category", "category", "id");
  }

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

Здесь все нам уже знакомо по предыдущей модели. Единственное: методу, устанавливающему связь, мы дали имя `cat`, чтобы избежать конфликтов с полем `category`.

Инструменты для работы со списками категорий и подкатегорий

Теперь можно приступить к созданию страниц для работы со списками категорий и подкатегорий.

Список категорий

Начнем с создания контроллера `CategoryController`, который будет заниматься выводом и правкой категорий. Дадим в консоли OpenServer команду:

```
php artisan make:controller CategoryController
```

Вывод списка категорий

Откроем файл с объявлением класса только что созданного контроллера `CategoryController` — это файл app\Http\Controllers\CategoryController.php. Исправим этот код согласно листингу 40.6.

Листинг 40.6

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

use Illuminate\Http\Request;
use App\Category;

class CategoryController extends Controller {
  public function index() {
    $cats = Category::orderBy("order", "desc")->orderBy("name")->get();
    return view("categories.index", ["cats" => $cats]);
  }
}
```

В методе-действии `index`, формирующем список категорий, мы сортируем их сначала по убыванию значения поля порядка (`order`), а потом — по названиям (`name`).

Создадим в папке resources\views папку categories, а в ней — файл шаблона index.blade.php с содержимым, представленным листингом 40.7.

Листинг 40.7

```
@extends("layouts.pc")

@push("head")
  <script src="/js/pc.js"></script>
@endpush

@section("title", "Категории")
@section("main")
  <h1>Список категорий</h1>
  <p><a href="{{ route('categories.create') }}">Создать категорию</a></p>
  <table class="list">
    <tr>
      <th>Порядок</th>
      <th>Название</th>
      <th>Слаг</th>
      <th colspan="2"> </th>
    <tr>
    @foreach ($cats as $cat)
      <tr>
        <td class="center">{{ $cat->order }}</td>
        <td>{{ $cat->name }}</td>
        <td>{{ $cat->slug }}</td>
        <td class="links">
          <a href="{{ action('CategoryController@input',
          ['category' => $cat->slug]) }}">Исправить</a>
        </td>
```

```
        <td class="links">
          <a href="{{ action('CategoryController@destroy',
          ['category' => $cat->slug]) }}" class="adel">Удалить</a>
        </td>
      </tr>
    @endforeach
  </table>
@endsection
```

Здесь мы помещаем непосредственно под заголовком **Список категорий** абзац с гиперссылкой, ведущей на страницу создания новой категории. В остальном все нам уже знакомо.

Автоматическое генерирование слагов

Чтобы упростить жизнь авторам, редакторам и администраторам нашего сайта, мы сделаем автоматическое генерирование слага на основе названия категории, подкатегории или статьи.

Слаг создается путем транслитерирования — преобразования букв кириллицы в буквы латиницы. Пробелы при этом преобразуются в другие символы, допустимые в интернет-адресах, — обычно в дефисы.

Мы напишем Web-сценарий, который при каждом изменении названия категории (подкатегории, статьи) будет генерировать на его основе слаг и подставлять его в соответствующее поле ввода. Поместим этот сценарий, код которого приведен в листинге 40.8, в тело обработчика события `load` окна, который хранится в файле public\js\pc.js.

Листинг 40.8

```
var arrChars = [
  ['а', 'a'], ['б', 'b'], ['в', 'v'], ['г', 'g'],
  ['д', 'd'],  ['е', 'e'], ['ё', 'yo'], ['ж', 'zh'], ['з', 'z'],
  ['и', 'i'], ['й', 'j'], ['к', 'k'], ['л', 'l'],
  ['м', 'm'],  ['н', 'n'], ['о', 'o'], ['п', 'p'],  ['р', 'r'],
  ['с', 's'], ['т', 't'], ['у', 'u'], ['ф', 'f'],
  ['х', 'h'],  ['ц', 'c'], ['ч', 'ch'],['ш', 'sh'], ['щ', 'ssh'],
  ['ъ', ''],  ['ы', 'y'], ['ь', ''],  ['э', 'e'], ['ю', 'yu'],
  ['я', 'ya'], [' ', '-']
];

function txtSourceInput() {
  var s = txtSource.value.toLowerCase();
  for (i in arrChars)
    s = s.replace(new RegExp(arrChars[i][0], "g"), arrChars[i][1]);
  txtDestination.value = s;
}
```

```
var txtSource = document.querySelector("input.source");
var txtDestination = document.querySelector("input.destination");
if ((txtSource) && (txtDestination))
  txtSource.addEventListener("input", txtSourceInput);
```

Здесь мы сначала объявим массив `arrChars`, задающий набор преобразуемых символов и символов, в которые они должны быть преобразованы. Каждый элемент этого массива также представляет собой массив с двумя элементами: преобразуемым символом и символом, в который он преобразуется.

Далее мы ищем на странице поля ввода с привязанными стилевыми классами `source` и `destination`. Из поля ввода со стилевым классом `source` будет извлечено исходное значение, которое после преобразования будет помещено в поле ввода с классом `destination`. Если оба поля ввода присутствуют на странице, мы привязываем к полю ввода со стилевым классом `source` обработчик события `input` — функцию `txtSourceInput`.

В теле этой функции мы перебираем все элементы массива `arrChars` и выполняем замену преобразуемых символов, встречающихся в исходной строке, на соответствующие им целевые символы. При этом на основе преобразуемого символа мы создаем регулярное выражение и задаем для него глобальный поиск.

Здесь мы также написали универсальный код, подходящий для формирования слагов как категорий, так и подкатегорий, и статей.

Правка списка категорий

Теперь дополним код контроллера `CategoryController` действиями, выполняющими создание, правку и удаление категорий. Добавленный нами код можно увидеть в листинге 40.9.

Листинг 40.9

```
. . .
use App\Http\Requests\CategoryRequest;

class CategoryController extends Controller {
  . . .
  public function input(Category $category) {
    if (!($category->id)) {
      $category->order = 0;
    }
    return view("categories.input", ["cat" => $category]);
  }

  public function save(CategoryRequest $request) {
    if ($request->has("id")) {
      $cat = Category::findOrFail($request->id);
      $cat->fill($request->all())->save();
```

```
      $s = " исправлена";
    } else {
      $cat = Category::create($request->all());
      $s = " создана";
    }
    return redirect()->action("CategoryController@index")
    ->with("status", "Категория " . $cat->name . $s);
  }

  public function destroy(Category $category) {
    $name = $category->name;
    $category->delete();
    return redirect()->action("CategoryController@index")
    ->with("status", "Категория " . $name . " удалена");
  }
}
```

Наиболее интересен для нас метод-действие `input`, выводящий страницу для добавления или правки категории. Он будет вызван при обращении по интернет-адресам вида **/categories/create** (создание категории) и **/categories/<*категория*>/edit** (правка). И при этом принимает в качестве параметра запись модели `Category`, найденную по взятому из интернет-адреса слагу.

Но в интернет-адресе первого вида слаг, указывающий на категорию, отсутствует! Что случится в таком случае? Laravel просто создаст новый объект класса модели `Category` — новую категорию — и передаст ее контроллеру. Так что нам самим создавать новую категорию в коде действия `input` не придется.

Однако нам понадобится проверить, новая ли категория была передана действию или уже существующая, и для новой категории занести в поле `order` значение по умолчанию — 0. Выполнить такую проверку мы можем, попытавшись получить значение поля `id` модели: если оно не содержит значения, значит, категория новая.

Не забудем создать класс запроса формы `CategoryRequest` для категории:

```
php artisan make:request CategoryRequest
```

Откроем файл app\Http\Requests\CategoryRequest.php, хранящий его код, и внесем в него набор правил для валидации и сообщения об ошибках (листинг 40.10).

Листинг 40.10

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

use Illuminate\Foundation\Http\FormRequest;

class CategoryRequest extends FormRequest {
  public function authorize() {
```

```
    return true;
  }

  public function rules() {
    return [
      'name' => 'required|max:30',
      'slug' => 'required|max:30',
      'order' => 'required|integer|max:255',
    ];
  }

  public function messages() {
    return [
      'name.required' => 'Введите название категории',
      'name.max' => 'Название категории должно быть не длиннее ' .
      '30 символов',
      'slug.required' => 'Введите слаг',
      'slug.max' => 'Слаг должен быть не длиннее 30 символов',
      'order.required' => 'Введите порядок',
      'order.integer' => 'Порядок должен быть целым числом',
      'order.max' => 'Порядок не должен превышать 255',
    ];
  }
}
```

И не забудем о шаблоне input.blade.php, формирующем страницу добавления и правки категории, — его код показан в листинге 40.11. Сохраним его в папке resources\views\categories.

Листинг 40.11

```
@extends("layouts.pc")

@push("head")
  <script src="/js/pc.js"></script>
@endpush

<?php $h = ($cat->id) ? $cat->name : "Добавление" ?>
@section("title", $h . " :: Категории")
@section("main")
  <h1>@if ($cat->id) Правка категории {{ $cat->name }}
  @else Добавление категории @endif</h1>
  <form action="{{ action('CategoryController@save') }}" method="POST">
    @if ($cat->id)
      {{ method_field('PUT') }}
      <input type="hidden" name="id" value="{{ old('id', $cat->id) }}">
    @endif
```

```
    {{ csrf_field() }}
    <label>Название</label>
    <input name="name" class="source"
    value="{{ old('name', $cat->name) }}" required>
    @include("common.errors", ["el" => "name"])
    <label>Слаг</label>
    <input name="slug" class="destination"
    value="{{ old('slug', $cat->slug) }}" required>
    @include("common.errors", ["el" => "slug"])
    <label>Порядок</label>
    <input type="number" name="order"
    value="{{ old('order', $cat->order) }}" required>
    @include("common.errors", ["el" => "order"])
    <input type="submit" value="Сохранить">
  </form>
  <p><a href="/categories">Список категорий</a></p>
@endsection
```

Здесь мы привязали к полю ввода названия стилевой класс `source`, а к полю ввода слага — стилевой класс `destination`. Это нужно для успешной работы сценария, генерирующего слаг на основе занесенного в поле названия. Также обратим внимание, что мы выводим разный текст в названии страницы и в заголовке первого уровня в зависимости от того, создается ли новая категория или правится уже имеющаяся.

Список подкатегорий

Инструменты для управления списком подкатегорий создаются аналогично таковым для работы со списком категорий. Вы можете изготовить их самостоятельно. Автор лишь опишет здесь некоторые специфические моменты.

Вывод списка подкатегорий

Подкатегории лучше всего вывести с указанием категорий, к которым они относятся. Так будущему администратору сайта будет удобнее с ними работать.

Код действия `index` контроллера `SubcategoryController`, который выводит этот список, показан в листинге 40.12.

Листинг 40.12

```
public function index() {
  $subcats = Subcategory::select("subcategories.*",
  "categories.name as catname")
  ->join("categories", "subcategories.category", "categories.id")
  ->orderBy("categories.order", "desc")->orderBy("categories.name")
  ->orderBy("order", "desc")->orderBy("name")->get();
  return view("subcategories.index", ["subcats" => $subcats]);
}
```

Ради повышения производительности для выборки списка подкатегорий мы создаем сложный запрос, в котором выполняем связывание таблиц `subcategories` и `categories`. Поле названия категории (`name`) будет доступно под псевдонимом `catname`.

Также обратим внимание, как сортируется список подкатегорий: сначала — по порядку категорий, потом — по их названиям и далее — по порядку и названиям подкатегорий. В результате администратор сразу увидит, какие подкатегории к каким категориям относятся и в каком порядке они будут следовать друг за другом.

Создание и правка подкатегорий

При создании и правке подкатегорий нам понадобится сформировать перечень категорий, на основе которого на странице создания и правки подкатегории будет создан раскрывающийся список категорий. Код действия `input` контроллера `SubcategoryController`, который выполняет в том числе и эту задачу, показан в листинге 40.13.

Листинг 40.13

```
public function input(Subcategory $subcategory) {
  if (!($subcategory->id)) {
    $subcategory->order = 0;
  }
  $cats = Category::orderBy("order", "desc")->orderBy("name")
  ->get("name", "id");
  return view("subcategories.input", ["subcat" => $subcategory,
  "cats" => $cats]);
}
```

А вот фрагмент кода шаблона resources\views\subcategories\input.blade.php, выводящий на экран раскрывающийся список категории:

```
<select name="category">
  @foreach ($cats as $cat)
    <option value="{{ $cat->id }}"
    @if (old('category', $subcat->category) == $cat->id)
    selected @endif>{{ $cat->name }}</option>
  @endforeach
</select>
```

Подобный код мы уже рассматривали в *главе 32*, когда говорили об обработке связей между таблицами.

Разграничение доступа

Чтобы указать фреймворку предоставить доступ к спискам категорий и подкатегорий только администраторам, мы можем создать две новые соответствующие политики. Но проще всего для этого применить написанную в *главе 39* политику `UserPolicy`. Так и поступим.

Откроем файл app\Providers\AuthServiceProvider.php, в котором хранится код класса провайдера `AuthServiceProvider`, и зарегистрируем в нем упомянутую ранее политику, связав ее с моделями `Category` и `Subcategory`. Для этого дополним ассоциативный массив, присваиваемый свойству `policies`, следующими элементами (добавленный код выделен полужирным шрифтом):

```
protected $policies = [
  'App\User' => 'App\Policies\UserPolicy',
  'App\Category' => 'App\Policies\UserPolicy',
  'App\Subcategory' => 'App\Policies\UserPolicy',
];
```

Откроем файл app\Http\Controllers\CategoryController.php с кодом класса контроллера `CategoryController` и добавим в этот класс конструктор:

```
public function __construct() {
  parent::__construct();
  $this->middleware("can:manipulate,App\Category")->except("view");
}
```

Мы разрешаем доступ ко всем действиям этого контроллера, кроме `view`, только администраторам. (Действие `view` будет выводить список подкатегорий, относящихся к выбранной посетителем категории, и, следовательно, должно быть доступно всем.)

После этого откроем файл app\Http\Controllers\SubcategoryController.php, где хранится код класса контроллера `SubcategoryController`, и также добавим в него конструктор:

```
public function __construct() {
  parent::__construct();
  $this->middleware("can:manipulate,App\Subcategory");
}
```

Здесь мы разрешаем доступ ко всем без исключения действиям контроллера только администраторам.

Панель навигации

Осталось вставить в панель навигации гиперссылки, указывающие на списки категорий и подкатегорий. Откроем шаблон resources\views\layouts\pc.blade.php и добавим в код, создающий панель навигации, фрагмент, показанный в листинге 40.14. Его можно поместить под кодом, создающим гиперссылку на страницу сведений о сайте.

Листинг 40.14

```
@can ("manipulate", "App\Category")
  <a href="/categories" @if ($p == "categories") class="active"
  @endif>Категории</a>
@endcan
```

```
@can ("manipulate", "App\Subcategory")
  <a href="/subcategories" @if ($p == "subcategories") class="active"
  @endif>Подкатегории</a>
@endcan
```

Вывод списков категорий и подкатегорий

Выведем теперь списки категорий и подкатегорий на общедоступных страницах нашего сайта.

На страницах «компьютерной» редакции сайта список категорий будет постоянно присутствовать в панели навигации. Здесь удобнее применить составитель, который будет формировать список категорий и который мы привяжем к родительскому шаблону resources\views\layouts\pc.blade.php. (О составителях рассказывалось в *главе 31*.)

Создадим в папке app\Http папку ViewComposers, а в ней — файл с объявлением класса `CategoryComposer`, который и станет нашим составителем. Его код показан в листинге 40.15.

Листинг 40.15

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

use Illuminate\View\View;
use App\Category;

class CategoryComposer {
  public function compose(View $view) {
    $view->with("ccats", Category::select("name", "slug")
    ->orderBy("order", "desc")->orderBy("name")->get());
  }
}
```

Здесь нет ничего примечательного. Мы формируем список категорий, извлекая поля названия и слага, и сохраняем его в переменной `ccats`, которая будет добавлена в контекст данных шаблона.

Теперь зарегистрируем составитель `CategoryComposer` во фреймворке. Откроем файл app\Providers\AppServiceProvider.php, где хранится код класса провайдера `AppServiceProvider`, и внесем в него нужные правки. Код исправленного провайдера показан в листинге 40.16.

Листинг 40.16

```
<?php
namespace App\Providers;
```

```
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider {
  public function boot() {
    View::composer("layouts.pc",
    "App\Http\ViewComposers\CategoryComposer");
  }

  public function register() {
  }
}
```

Гиперссылки панели навигации, соответствующие категориям, будут иметь интернет-адреса вида **view/<*категория*>**, где *категория* представлена слагом. При щелчке на такой гиперссылке выполнится действие `view` контроллера `CategoryController`, которое сформирует список подкатегорий, относящихся к этой категории. Давайте откроем файл app\Http\Controllers\CategoryController.php, в котором сохранен код этого контроллера, и вставим в него объявление метода-действия `view` (листинг 40.17).

Листинг 40.17

```
. . .
use App\Subcategory;

class CategoryController extends Controller {
  . . .
  public function view(Category $category) {
    $subcats = Subcategory::select("name", "slug")
    ->where("category", $category->id)
    ->orderBy("order", "desc")->orderBy("name")->get();
    return view("categories.view", ["subcats" => $subcats,
    "currentCat" => $category]);
  }
}
```

Мы указываем выбрать только подкатегории, относящиеся к выбранной посетителем категории, извлекаем поля названия и слага подкатегории. Получившийся список мы помещаем в контекст данных шаблона resources\views\categories\view.blade.php под именем `subcats`, наряду с выбранной посетителем категорией (она будет доступна в шаблоне под именем `currentCat` и понадобится нам, чтобы вывести на странице наименование категории и правильно сгенерировать панель навигации).

Настала пора шаблона resources\views\layouts\pc.blade.php, задающего разметку для страниц «компьютерной» редакции сайта. Откроем его и вставим в код, что создает панель навигации, фрагмент, который в листинге 40.18 выделен полужирным шрифтом.

Листинг 40.18

```
<nav>
  <?php $p = request()->path(); ?>
  <a href="/" @if ($p == "/") class="active" @endif>Главная</a>
  @foreach ($ccats as $ccat)
    <?php $h = "view/" . $ccat->slug; ?>
    <a href="/{{ $h }}" @if ($p == $h) class="active"
    @endif>{{ $ccat->name}}</a>
    @if ((isset($currentCat)) && ($ccat->slug == $currentCat->slug))
      <div>
        @foreach ($subcats as $subcat)
          <?php $h = "view/" . $ccat->slug . "/" . $subcat->slug; ?>
          <a href="/{{ $h }}" @if ($p == $h) class="active"
          @endif>{{ $subcat->name}}</a>
        @endforeach
      </div>
    @endif
  @endforeach
  . . .
</nav>
```

Этот код перебирает категории, чей список сгенерирован составителем `CategoryComposer` и хранится в переменной `ccats`, и на основе каждой категории создает гиперссылку. Затем выполняется проверка, присутствует ли в контексте данных шаблона переменная `currentCat`, хранящая выбранную посетителем категорию (то есть выбрал ли посетитель какую-либо категорию), и, если присутствует, совпадает ли слаг выбранной категории со слагом очередной категории, взятой из списка (то есть является ли очередная категория той, которую выбрал посетитель). Если эти условия выполняются, в панели навигации под гиперссылкой категории будет выведен список относящихся к ней подкатегорий, взятый из переменной `subcats`.

В приведенном в листинге 40.18 коде для проверки существования переменной `currentCat` применяется функция PHP `isset`. Она принимает в качестве аргумента переменную и возвращает `true`, если эта переменная объявлена и существует, и `false` в противном случае.

Так, с «компьютерными» страницами мы закончили. Позаботимся о поклонниках мобильных устройств, внеся правки в шаблоны «мобильной» редакции сайта.

На главной странице этой редакции нам нужно вывести гиперссылки, указывающие на категории. Следовательно, сначала нужно исправить код метода-действия `index` контроллера `MainController`, добавив в него код, создающий список категорий. Откроем файл app\Http\Controllers\MainController.php и сделаем это (листинг 40.19).

Листинг 40.19

```
. . .
use App\Category;

class MainController extends Controller {
  public function index() {
    if ($this->isMobile) {
      $cats = Category::orderBy("order", "desc")->orderBy("name")->get();
    } else {
      $cats = null;
    }
    return view("main", ["cats" => $cats]);
  }
  . . .
}
```

Отметим, что список категорий формируется только в случае обращения с мобильного устройства. Если обращение будет выполнено с традиционного компьютера, формировать его ни к чему, — мы лишь займем бесполезными данными оперативную память.

Откроем шаблон главной страницы resources\views\main.blade.php и исправим код, создающий список в конце страницы (в листинге 40.20 показано то, что должно получиться в результате).

Листинг 40.20

```
@if (request()->route()->getController()->isMobile)
  <ul>
    @foreach ($cats as $cat)
      <li><a href="/view/{{ $cat->slug }}">{{ $cat->name}}</a></li>
    @endforeach
    <li><a href="/policy">Политика публикации статей</a></li>
    <li><a href="/about">Сведения о сайте</a></li>
  </ul>
@endif
```

И, наконец, в папке resources\views\categories создадим файл шаблона view.blade.php, который сгенерирует страницу, выводящуюся на экран после выбора посетителем категории — неважно, в панели навигации или в списке на главной странице. Код этого шаблона показан в листинге 40.21.

Листинг 40.21

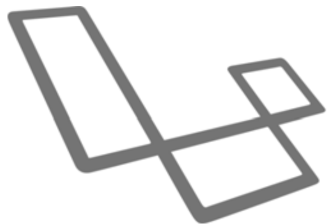
```
@extends(request()->route()->getController()->parentView)

@section("title", $currentCat->name)
@section("main")
```

```
  @if (request()->route()->getController()->isMobile)
    <p><a href="/">Главная</a></p>
  @endif
  <h1>{{ $currentCat->name }}</h1>
  @if (request()->route()->getController()->isMobile)
    <ul>
      @foreach ($subcats as $subcat)
        <li><a href="/view/{{ $currentCat->slug }}/{{ $subcat->slug}}">
        {{ $subcat->name}}</a></li>
      @endforeach
    </ul>
  @endif
@endsection
```

При обращении с традиционного компьютера на этой странице будет выведен пока только заголовок с названием выбранной категории. (Мы еще доработаем эту страницу в *главе 41*.) А в случае обращения с мобильного устройства на ней также будут присутствовать гиперссылка на главную страницу и список подкатегорий.

Войдем на сайт от имени пользователя с ролью администратора, добавим несколько категорий и подкатегорий, проверим, работают ли функции правки и удаления. Посмотрим, правильно ли выводится панель навигации. Напоследок проверим, как отображаются страницы «мобильной» редакции сайта.



ГЛАВА 41

Статьи. Поддержка BBCode

В этой главе мы займемся статьями — тем, ради чего и создается этот сайт. Мы реализуем вывод на экран перечня статей, относящихся к выбранной посетителем подкатегории, вывод последних пяти статей, относящихся к выбранной категории, вывод содержимого статьи, возможность добавления, правки и удаления статей и поиск статей по заданному ключевому слову.

Маршруты

Откроем файл routes\web.php и добавим к его содержимому фрагмент кода, показанный в листинге 41.1.

Листинг 41.1

```
Route::get("/view/{category}/{subcategory}/{article}",
"ArticleController@view");
Route::get("/search", "ArticleController@search");

Route::get("/articles/{category}/{subcategory}/create",
"ArticleController@input")->name("articles.create");
Route::post("/articles", "ArticleController@save");
Route::get("/articles/{category}/{subcategory}/{article}/edit",
"ArticleController@input");
Route::put("/articles", "ArticleController@save");
Route::get("/articles/{category}/{subcategory}/{article}/delete",
"ArticleController@destroy");
```

Прежде всего, мы написали маршрут, связывающий интернет-адрес вида **/view/<*категория*>/<*подкатегория*>/<*статья*>** с действием `view` контроллера `ArticleController`. Здесь мы продолжаем политику выстраивания иерархии категорий, подкатегорий и статей. Еще мы связали интернет-адрес **/search** с действием `search` того же контроллера, которое будет выполнять поиск статей и выводить страницу с его результатами.

Миграции и модели

Здесь нам понадобится создать новую таблицу, в которой будут храниться статьи, и исправить модели пользователей и подкатегорий.

Миграция и модель списка статей

Все статьи, что опубликованы на нашем сайте, будут храниться в таблице `articles` базы данных. Ее структуру можно увидеть в табл. 41.1 (служебные поля не показаны).

Таблица 41.1. *Структура таблицы* `articles`

Имя поля	Тип поля	Параметры поля	Назначение поля
`title`	`VARCHAR`	100 символов, обязательное, уникальный индекс	Заголовок статьи
`slug`	`VARCHAR`	100 символов, обязательное, уникальный индекс	Слаг статьи
`description`	`VARCHAR`	255 символов, обязательное	Анонс статьи
`content`	`TEXT`	Обязательное	Содержимое статьи
`subcategory`	`INT`	Беззнаковое, индекс	Подкатегория
`author`	`INT`	Беззнаковое, индекс	Автор

Поле `description` будет содержать краткий анонс статьи, выводящийся на странице списка статей. Если автор статьи не заполнил это поле, в него будут автоматически подставлены первые 254 символа содержимого статьи и символ многоточия.

Позднее мы создадим две связи «один-ко-многим»:

- между таблицами `subcategories` (которая станет первичной) и `articles` (она станет вторичной). Поле `subcategory` второй таблицы станет внешним ключом;
- между таблицами `users` (первичная) и `articles` (вторичная). Внешним ключом станет поле `author` второй таблицы.

Создадим миграцию для таблицы `articles`, для чего запустим консоль OpenServer, перейдем в папку проекта и выполним такую команду:

```
php artisan make:migration create_articles_table --create=articles
```

Найдем в папке database\migrations файл *<дата и время>*_create_articles_table.php, в котором хранится объявление класса миграции `CreateArticlesTable`, откроем его и исправим код метода `up` этого класса (листинг 41.2).

Листинг 41.2

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

```
    $table->string('title', 100)->unique();
    $table->string('slug', 100)->unique();
    $table->string('description', 255);
    $table->text('content');
    $table->integer('subcategory')->unsigned();
    $table->integer('author')->unsigned();
    $table->timestamps();
    $table->foreign('subcategory')->references('id')->on('subcategories')
    ->onDelete('restrict')->onUpdate('restrict');
    $table->foreign('author')->references('id')->on('users')
    ->onDelete('restrict')->onUpdate('restrict');
  });
}
```

Выполним миграцию:

```
php artisan migrate
```

Создадим модель `Article`:

```
php artisan make:model Article
```

Откроем только что созданный файл app\Article.php с классом этой модели и исправим код этого класса, как показано в листинге 41.3.

Листинг 41.3

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model {
  protected $fillable = ["title", "slug", "description", "content",
  "subcategory", "author"];

  public function subcat() {
    return $this->belongsTo("App\Subcategory", "subcategory", "id");
  }

  public function written_by() {
    return $this->belongsTo("App\User", "author", "id");
  }

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

Для метода, создающего связь с моделью `Subcategory`, мы указали имя `subcat`, а для метода, что устанавливает связь с моделью `User`, — `written_by`. Если мы укажем для этих методов имена `subcategory` и `author`, возникнут конфликты с одноименными полями таблицы. Остальное нам уже знакомо.

Модели списков подкатегорий и пользователей

Поскольку мы установили обратную связь между моделями `Subcategory` и `Article`, то должны также задать и прямую связь. Откроем файл app\Subcategory.php, где хранится код класса `Subcategory`, и добавим в него следующий метод, создающий эту связь:

```
public function articles() {
  return $this->hasMany("App\Article", "subcategory", "id");
}
```

После чего откроем файл app\User.php, хранящий код класса `User`, в который добавим такой метод:

```
public function articles() {
  return $this->hasMany("App\Article", "author", "id");
}
```

Разграничение доступа

Ограничение доступа к страницам для работы со статьями удобнее реализовать прямо сейчас, сразу после создания моделей. Создавать новые статьи у нас сможет любой зарегистрированный пользователь, а править и удалять — либо их автор, либо любой пользователь с ролью редактора или администратора.

Ограничением доступа к статьям будет управлять политика `ArticlePolicy`. Создадим ее:

```
php artisan make:policy ArticlePolicy
```

Откроем только что созданный файл app\Policies\ArticlePolicy.php, где сохранен код вновь созданного класса, и внесем в него необходимые правки (листинг 41.4).

Листинг 41.4

```
<?php
namespace App\Policies;

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

class ArticlePolicy {
  use HandlesAuthorization;
```

```
  public function manipulate(User $user, Article $article) {
    return (($user->role == "a") && ($user->id == $article->author) ||
    ($user->role != "a"));
  }
}
```

Операция `manipulate`, заданная одноименным методом, будет соответствовать правке и удалению статьи.

Откроем файл app\Providers\AuthServiceProvider.php, в котором хранится код класса провайдера `AuthServiceProvider`, и зарегистрируем в нем политику `ArticlePolicy`, связав ее с моделью `Article`. Сделаем это, добавив в ассоциативный массив, присваиваемый свойству `policies`, такой элемент:

```
protected $policies = [
  . . .
  'App\Article' => 'App\Policies\ArticlePolicy',
];
```

Вывод списка статей

Теперь займемся кодом, который будет выводить списки статей и содержимое статьи, выбранной посетителем.

Вывод списка статей, относящихся к выбранной подкатегории

Начнем с вывода списка статей, относящихся к выбранной посетителем подкатегории. Для этого сначала сгенерируем контроллер `ArticleController`, отдав в консоли OpenServer команду:

```
php artisan make:controller ArticleController
```

Откроем файл app\Http\Controllers\ArticleController.php, в котором хранится код только что сгенерированного класса контроллера. Исправим его, как показано в листинге 41.5.

Листинг 41.5

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

use Illuminate\Http\Request;
use App\Article;
use App\Category;
use App\Subcategory;
```

```
class ArticleController extends Controller {
  public function __construct() {
    parent::__construct();
    $this->middleware("auth")->except(["index", "view", "search"]);
  }

  public function index(Category $category, Subcategory $subcategory) {
    $articles = Article::select("articles.*", "users.name as authorname")
    ->join("users", "articles.author", "users.id")
    ->where("articles.subcategory", $subcategory->id)
    ->latest("articles.created_at")->orderBy("articles.title")
    ->paginate(1);
    if ($this->isMobile) {
      $subcats = null;
    } else {
      $subcats = Subcategory::select("name", "slug")
      ->where("category", $category->id)
      ->orderBy("order", "desc")->orderBy("name")->get();
    }
    return view("articles.index", ["articles" => $articles,
    "subcats" => $subcats, "currentSubcat" => $subcategory,
    "currentCat" => $category]);
  }
}
```

Первое, что мы сделали, — добавили в класс конструктор. В нем мы указали для всех действий, кроме `index` (выводящего список статей), `view` (отображающего выбранную статью) и `search` (производящего поиск статей), посредник `auth`, тем самым сделав их недоступными для гостей.

Метод-действие `index` у нас принимает в качестве параметров категорию и подкатегорию, которые выбрал посетитель. (Вспомним, что ранее в маршрутах мы связали интернет-адрес **/view/<*категория*>/<*подкатегория*>** как раз с этим действием.) Таким образом, мы сможем без проблем получить список статей, относящихся к выбранной посетителем подкатегории.

В методе-действии `index` мы формируем список статей, относящихся к выбранной категории. Помимо всех полей таблицы `articles`, мы также извлекаем значение поля `name` таблицы `users` (то есть имя пользователя — автора статьи), предварительно связав эти таблицы. Конечно, для получения имени автора статьи мы можем воспользоваться средствами доступа к связанным записям, описанными в *главе 30*, но те средства довольно медленные.

На основе полученного списка статей мы создаем пагинатор (см. *главу 30*). И для отладочных целей указываем количество записей, выводящихся на одной странице, равным 1, — благодаря этому мы сможем проверить работу пагинатора, добавив всего лишь пару статей.

Если обращение выполняется с традиционного компьютера, мы также извлекаем список подкатегорий, относящихся к выбранной категории. Это нужно для того, чтобы вывести этот список в панели навигации под выбранной категорией.

Все полученные данные, а также выбранные категорию и подкатегорию мы передаем шаблону resources\views\articles\index.blade.php, который создаст страницу списка статей. Создадим в папке resources\views папку articles и сохраним в ней файл index.blade.php с кодом, показанным в листинге 41.6.

Листинг 41.6

```
@extends(request()->route()->getController()->parentView)

@push("head")
  <script src="/js/pc.js"></script>
@endpush

@section("title", $currentSubcat->name . " :: " . $currentCat->name)
@section("main")
  @if (request()->route()->getController()->isMobile)
    <p><a href="/">Главная</a>
    &lt;&lt; <a href="{{ action('CategoryController@view',
    ['category' => $currentCat->slug]) }}">
    {{ $currentCat->name }}</a></p>
  @endif
  <h1>{{ $currentSubcat->name }}</h1>
  @if (auth()->check())
    <p><a href="{{ route('articles.create',
    ['category' => $currentCat->slug,
    'subcategory' => $currentSubcat->slug, 'page' => 1]) }}">
    Создать статью</a></p>
  @endif
  @foreach ($articles as $article)
    <article>
      <?php
        $params = [
          "category" => $currentCat->slug,
          "subcategory" => $currentSubcat->slug,
          "article" => $article->slug,
          "page" => (request()->has("page")) ?
          request()->input("page") : 1,
        ];
      ?>
      <h2><a href="{{ action('ArticleController@view', $params) }}">
      {{ $article->title }}</a></h2>
      <div class="author-created">{{ $article->authorname }} ||
      {{ date_format($article->created_at, "j.m.Y G:i") }}</div>
```

```
      <div>{{ $article->description }}</div>
      @can("manipulate", $article)
        <div class="links">
          <a href=" {{ action('ArticleController@input', $params) }}">
          Исправить</a>
          <a href=" {{ action('ArticleController@destroy', $params) }}"
          class="adel">Удалить</a>
        </div>
      @endcan
    </article>
  @endforeach
  {{ $articles->links() }}
@endsection
```

Если обращение выполняется с мобильного устройства, вверху страницы мы выводим гиперссылки, указывающие на главную страницу и на страницу со списком подкатегорий.

Теперь обратим внимание на PHP-код, создающий ассоциативный массив `params`:

```
$params = [
  "category" => $currentCat->slug,
  "subcategory" => $currentSubcat->slug,
  "article" => $article->slug,
  "page" => (request()->has("page")) ?
  request()->input("page") : 1,
];
```

Он содержит параметры, на основе значений которых будут сгенерированы интернет-адреса, указывающие на действия `view`, `input` и `destroy`. В число этих параметров входят слаги выбранных категории, подкатегории, очередной статьи и номер текущей страницы пагинатора. Слаги понадобятся для корректного генерирования интернет-адресов, а указание номера страницы — для возврата на ту страницу списка статей, с которой произошел переход на страницу с выбранной статьей.

Для правильного форматирования значения даты и времени создания статьи перед выводом мы применили функцию `date_format`, встроенную в PHP. Вот формат ее вызова:

```
date_format(<форматируемое значение>, <формат>)
```

В качестве результата она возвращает отформатированное значение даты и времени в виде строки. Формат, использованный нами: `"j.m.Y G:i"` — выдает дату и время в привычном нам виде *<число>.<номер месяца>.<год> <часы>:<минуты>*.

Разумеется, если текущий пользователь имеет право создавать, править и удалять статьи, мы выводим соответствующие гиперссылки. В интернет-адрес гиперссылки на страницу создания статьи мы помещаем GET-параметр `page`, то есть номер текущей страницы пагинатора, со значением 1, благодаря чему после добавления новой статьи произойдет перенаправление на самую первую страницу списка статей.

Теперь немного подправим шаблон пагинатора, который хранится в файле resources\views\vendor\pagination\default.blade.php. Исправленный код можно увидеть в листинге 41.7.

Листинг 41.7

```
@if ($paginator->hasPages())
  <div class="pagination">
    @foreach ($elements as $element)
      @if (is_string($element))
        {{ $element }}  
      @endif

      @if (is_array($element))
        @foreach ($element as $page => $url)
          @if ($page == $paginator->currentPage())
            {{ $page }}  
          @else
            <a href="{{ $url }}">{{ $page }}</a>  
          @endif
        @endforeach
      @endif
    @endforeach
  </div>
@endif
```

Здесь сначала выполняется проверка, не единственная ли это страница с записями (если это единственная страница, выводить пагинатор бессмысленно). Далее перебираются элементы пагинатора и выполняется проверка их типа:

- если очередной элемент — строка, выполняется вывод этого элемента и двух неразрывных пробелов за ним.

  Для определения, является ли элемент массива строкой, здесь применена функция `is_string` PHP. Она принимает в качестве единственного аргумента значение и возвращает `true`, если оно имеет строковый тип, и `false` в противном случае;
- если очередной элемент — массив, предполагается, что это ассоциативный массив гиперссылок, указывающих на отдельные страницы пагинатора. Если очередной элемент указывает на текущую страницу, выводится лишь ее номер, в противном случае формируется гиперссылка. В любом случае после номера страницы или гиперссылки помещаются два неразрывных пробела.

  Определить, является ли элемент массива массивом, поможет функция `is_array`. Она принимает в качестве единственного аргумента значение и возвращает `true`, если это массив, и `false` в противном случае.

Остальные файлы шаблонов, хранящиеся в папке resources\views\vendor\pagination, нам не нужны, и их можно удалить.

Осталось лишь добавить в таблицы стилей несколько новых стилей, которые зададут оформление списка статей. Эти стили, вместе с указанием на таблицу стилей, куда они должны быть помещены, приведены в листинге 41.8.

Листинг 41.8

```
/* basic.css */
article {            border: 1px solid grey;
                     margin: 10px 0px;
                     padding: 10px; }
article h2 {         margin-top: 0px; }
article h2:before {  counter-increment: none;
                     content: none; }
.author-created {    font-weight: bold;
                     margin-bottom: 20px; }
.pagination {        margin-top: 10px;
                     text-align: center; }

/* pc.css */
article div.links {  text-align: right; }
.author-created {    font-size: larger;
                     text-align: right; }
```

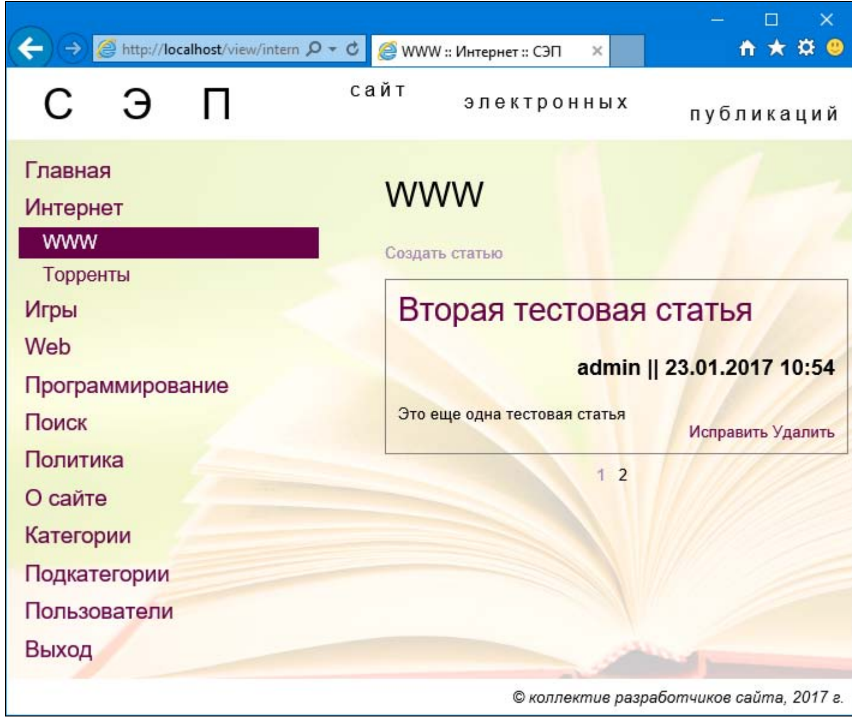


Рис. 41.1. Страница списка статей

В таблицу стилей pc.css мы вынесли стили, задающие оформление для блока с гиперссылками на административные страницы и устанавливающие выравнивание по правому краю и полужирное начертание для имени автора и даты создания статьи. На страницах «компьютерной» редакции сайта имеет смысл выделить эти сведения, в то время как на страницах «мобильной» редакции они и так будут достаточно заметными.

На рис. 41.1 показана страница списка статей. Как уже говорилось ранее, для целей отладки мы задали количество позиций, помещающихся на одной странице пагинатора, равному 1.

Вывод списка последних пяти статей, относящихся к выбранной категории

Следующий шаг — вывод списка последних пяти статей, относящихся к выбранной посетителем категории, на страницах редакции сайта, которая предназначена для традиционных компьютеров. Займется этим действие `view` контроллера `CategoryController`, написанное нами в *главе 40*.

Откроем файл app\Http\Controllers\CategoryController.php и исправим код метода-действия `view` соответственно листингу 41.9.

Листинг 41.9

```
public function view(Category $category) {
  $subcats = Subcategory::select("name", "slug")
  ->where("category", $category->id)
  ->orderBy("order", "desc")->orderBy("name")->get();
  if ($this->isMobile) {
    $articles = null;
  } else {
    $articles = Article::select("articles.*", "users.name as authorname",
    "subcategories.name as subcat", "categories.name as cat",
    "subcategories.slug as subcatslug", "categories.slug as catslug")
    ->join("users", "articles.author", "users.id")
    ->join("subcategories", "articles.subcategory", "subcategories.id")
    ->join("categories", "subcategories.category", "categories.id")
    ->where("subcategories.category", $category->id)
    ->latest("articles.created_at")->orderBy("articles.title")->limit(5)
    ->get();
  }
  return view("categories.view", ["subcats" => $subcats,
  "currentCat" => $category, "articles" => $articles]);
}
```

Здесь мы используем запрос, аналогичный представленному в листинге 41.5, и не забываем ограничить количество извлекаемых записей пятью. А также извлекаем названия категории и подкатегории, к которым относится каждая статья.

Осталось дополнить код шаблона resources\views\categories\view.blade.php фрагментом, представленным в листинге 41.10.

Листинг 41.10

```
@if (request()->route()->getController()->isMobile)
  . . .
@else
  @foreach ($articles as $article)
    <article>
      <p>{{ $article->cat }} - {{ $article->subcat }}</p>
      <h2><a href="{{ action('ArticleController@view',
      ['category' => $article->catslug,
      'subcategory' => $article->subcatslug,
      'article' => $article->slug]) }}">{{ $article->title }}</a></h2>
      <div class="author-created">{{ $article->authorname}} ||
      {{ date_format($article->created_at, "j.m.Y G:i") }}</div>
      <div>{{ $article->description}}</div>
    </article>
  @endforeach
@endif
. . .
```

Здесь мы выводим список статей, если обращение к сайту было выполнено с традиционного компьютера.

Поиск статей

Поиск статей у нас будет выполнять действие `search` того же контроллера `ArticleController`. Сразу же добавим в панель навигации пункт, ведущий на это действие, а также поместим соответствующую гиперссылку на главную страницу «мобильной» редакции сайта.

Откроем шаблон resources\views\layouts\pc.blade.php, найдем код, создающий панель навигации, и добавим в него строку:

```
<a href="/search" @if ($p == "search") class="active" @endif>Поиск</a>
```

Ее можно вставить после кода, создающего пункты, которые указывают на категории.

После этого откроем шаблон resources\views\main.blade.php, найдем код, создающий маркированный список, и вставим в него строку (также после пунктов, указывающих на категории):

```
<li><a href="/search">Поиск</a></li>
```

И со спокойной совестью приступим к работе над действием `search` контроллера `ArticleController` (его объявление хранится в файле app\Http\Controllers\ArticleController.php). Код, который мы напишем, приведен в листинге 41.11.

Листинг 41.11

```
public function search() {
  if (request()->has("search")) {
    $s = request()->input("search");
    $articles = Article::select("articles.*", "users.name as authorname",
    "subcategories.name as subcat", "categories.name as cat",
    "subcategories.slug as subcatslug", "categories.slug as catslug")
    ->join("users", "articles.author", "users.id")
    ->join("subcategories", "articles.subcategory", "subcategories.id")
    ->join("categories", "subcategories.category", "categories.id")
    ->where("articles.title", "like", "%" . $s . "%")
    ->orWhere("articles.content", "like", "%" . $s . "%")
    ->latest("articles.created_at")->orderBy("articles.title")
    ->paginate(1);
    $articles->appends("search", $s);
  } else {
    $articles = null;
  }
  return view("articles.search", ["articles" => $articles]);
}
```

Сначала мы проверяем, был ли передан в составе запроса GET-параметр `search`, значением которого является ключевое слово для поиска. Если он был передан, мы выполняем поиск статей, в противном случае не делаем этого.

Запрос, который мы сформируем для поиска статей, аналогичен тому, что приведен в листинге 41.5. Мы отбираем только те статьи, заголовки (поле `title`) и содержание (поле `content`) которых содержат введенное посетителем ключевое слово. Для этого мы применяем методы `where` и `orWhere`, описанные в *главе 30*. Еще мы извлекаем слаги категорий и подкатегорий, к которым относится каждая статья, — это нужно для правильного формирования интернет-адресов, указывающих на статью.

Поскольку список найденных статей может быть очень велик, на основе полученного списка статей мы создаем пагинатор. «Размер» его страницы мы также указываем равным 1 для облегчения отладки.

Гиперссылки пагинатора, указывающие на отдельные его статьи, должны включать в свой состав GET-параметр `search` с ключевым словом, — в противном случае при переключении на другую страницу пагинатора поиск не будет выполнен, и посетитель увидит пустую страницу. Такой параметр к интернет-адресу, на основе которого будут формироваться интернет-адреса этих гиперссылок, мы добавляем вызовом метода `appends` (см. *главу 30*).

Код шаблона resources\views\articles\search.blade.php, создающего страницу с результатами поиска, представлен в листинге 41.12. Он похож на код аналогичного шаблона index.blade.php, находящегося в той же папке и выводящего страницу со списком статей, за исключением некоторых важных моментов.

Листинг 41.12

```
@extends(request()->route()->getController()->parentView)

@section("title", "Поиск")
@section("main")
  @if (request()->route()->getController()->isMobile)
    <p><a href="/">Главная</a></p>
  @endif
  <h1>Поиск статей</h1>
  <form action="/search" method="get">
    <input type="text" name="search"
    value="{{ request()->input('search') }}" class="wide">
    <input type="submit" value="Найти">
  </form>
  @if ($articles)
    @foreach ($articles as $article)
      <article>
        <?php
          $params = [
            "category" => $article->catslug,
            "subcategory" => $article->subcatslug,
            "article" => $article->slug,
            "search" => request()->input("search"),
            "page" => (request()->has("page")) ?
            request()->input("page") : 1,
          ];
        ?>
        <p>{{ $article->cat }} - {{ $article->subcat }}</p>
        <h2><a href="{{ action('ArticleController@view', $params) }}">
        {{ $article->title }}</a></h2>
        <div class="author-created">{{ $article->authorname }} ||
        {{ date_format($article->created_at, "j.m.Y G:i") }}</div>
        <div>{{ $article->description }}</div>
        @can("manipulate", $article)
          <div class="links">
            <a href="{{ action('ArticleController@input', $params) }}">
            Исправить</a>
            <a href="{{ action('ArticleController@destroy', $params) }}"
            class="adel">Удалить</a>
          </div>
        @endcan
      </article>
    @endforeach
    {{ $articles->links() }}
  @endif
@endsection
```

Во-первых, не забываем вывести форму поиска. Она обычно находится в самом верху страницы — там-то мы ее и поместим.

Во-вторых, мы предоставляем возможность править и удалять страницы прямо отсюда — со страницы поиска. Так мы облегчим работу зарегистрированным пользователям нашего сайта.

В-третьих, в составе интернет-адресов гиперссылок, ведущих на страницы просмотра, правки и удаления статьи, мы дополнительно передаем GET-параметр `search` с введенным посетителем ключевым словом. Это необходимо, чтобы реализовать возврат на страницу с выведенными на ней результатами последнего поиска.

Теперь растянем поле для занесения ключевого слова на всю ширину родителя (формы). Мы уже привязали к этому полю стилевой класс `wide`, так что теперь останется лишь добавить в таблицу стилей basic.css следующий стиль:

```
.wide { width: calc(100% - 20px); }
```

Вывод статьи

При реализации вывода выбранной посетителем статьи нам понадобится решить две проблемы.

1. Обеспечить поддержку тегов BBCode, то есть их корректное преобразование в соответствующие HTML-теги.
2. Поскольку на страницу статьи посетитель сможет попасть со страницы списка статей, относящихся в выбранной подкатегории, со страницы категории, на которой выводится список последних пяти созданных статей, и со страницы результатов поиска, также реализовать возврат на страницу, с которой был выполнен переход.

Мы будем решать эти проблемы по очереди.

Форматирование текста статей. BBCode

Итак, для форматирования содержания статей мы применим так называемые теги *BBCode* (Bulletin Board Code, код доски объявлений). Они аналогичны тегам HTML, но формируются не символами < и >, а квадратными скобками и поддерживают достаточно ограниченный набор возможностей форматирования, которого, тем не менее, хватает для публикации даже больших и сложно структурированных текстов.

Набор тегов BBCode, поддерживаемых нашим сайтом

Для начала давайте определимся с набором тегов BBCode, которые будет поддерживать наш сайт:

- `[b]`*<текст>*`[/b]` — полужирный *текст*;
- `[i]`*<текст>*`[/i]` — курсивный *текст*;

- `[url=<интернет-адрес>]<текст>[/url]` — гиперссылка с заданными *текстом* и *интернет-адресом*;
- `[center]<текст>[/center]` — центрированный *текст*;
- `[right]<текст>[/right]` — *текст*, выровненный по правому краю;
- `[h<уровень>]<текст>[/h<уровень>]` — заголовок заданного *уровня* (доступны значения от 1 до 5);
- `[img]<интернет-адрес>[/img]` — изображение с заданным *интернет-адресом*;
- `[sign]<текст>[/sign]` — подпись к изображению;
- `[code]<текст>[/code]` — текст фиксированного форматирования;
- `[audio]<интернет-адрес>[/audio]` — аудиоролик;
- `[video]<интернет-адрес>[/video]` — видеоролик;
- `[spoiler=<заголовок>]<содержимое>[/spoiler]` — спойлер;
- `[lightbox=<интернет-адрес миниатюры>]<интернет-адрес основного изображения>[/lightbox]` — лайтбокс.

Этих тегов, на взгляд автора, хватит для форматирования статей любого размера. Если же возникнет необходимость расширить набор тегов, это можно будет сделать в любой момент.

Собственно форматирование текстов статей

Код, преобразующий теги BBCode в HTML-теги, мы поместим в класс модели `Article`, добавив в него метод `getFormattedContentAttribute` и создав тем самым доступное только для считывания свойство `formatted_content`.

Откроем файл app\Article.php и вставим в объявление класса код, показанный в листинге 41.13. Он довольно объемный, но бóльшая часть его выражений практически однотипна.

Листинг 41.13

```
public function getFormattedContentAttribute() {
  $text = htmlentities($this->attributes["content"]);
  $text = preg_replace("/\n(.*?)\r/si", "<p>\\1</p>", "\n". $text .
  "\r");
  $text = preg_replace("/\[b\](.*?)\[\/b\]/si", "<strong>\\1</strong>",
  $text);
  $text = preg_replace("/\[i\](.*?)\[\/i\]/si", "<em>\\1</em>", $text);
  $text = preg_replace("/<p>\[center\](.*?)\[\/center\]<\/p>/si",
  "<p class='center'>\\1</p>", $text);
  $text = preg_replace("/<p>\[right\](.*?)\[\/right\]<\/p>/si",
  "<p class='right'>\\1</p>", $text);
  $text = preg_replace("/<p>\[h1\](.*?)\[\/h1\]<\/p>/si",
  "<h2>\\1</h2>", $text);
```

```
  $text = preg_replace("/<p>\[h2\](.*?)\[\/h2\]<\/p>/si",
  "<h3>\\1</h3>", $text);
  $text = preg_replace("/<p>\[h3\](.*?)\[\/h3\]<\/p>/si",
  "<h4>\\1</h4>", $text);
  $text = preg_replace("/<p>\[h4\](.*?)\[\/h4\]<\/p>/si",
  "<h5>\\1</h5>", $text);
  $text = preg_replace("/<p>\[h5\](.*?)\[\/h5\]<\/p>/si",
  "<h6>\\1</h6>", $text);
  $text = preg_replace("/<p>\[img\](.*?)\[\/img\]<\/p>/si",
  "<figure><img src='\\1'></figure>", $text);
  $text = preg_replace("/\[img\](.*?)\[\/img\]/si", "<img src='\\1'>",
  $text);
  $text = preg_replace("/<p>\[sign\](.*?)\[\/sign\]<\/p>/si",
  "<div class='sign'>\\1</div>", $text);
  $text = preg_replace("/\[url=(.*?)\](.*?)\[\/url\]/si",
  "<a href='\\1'>\\2</a>", $text);
  $text = preg_replace("/<p>\[audio\](.*?)\[\/audio\]<\/p>/si",
  "<figure><audio controls preload='metadata' src='\\1'>" .
  "</audio></figure>", $text);
  $text = preg_replace("/<p>\[video\](.*?)\[\/video\]<\/p>/si",
  "<figure><video controls preload='metadata' src='\\1'>" .
  "</video></figure>", $text);
  $text = preg_replace("/<p>\[code\](.*?)\[\/code\]<\/p>/si",
  "<pre><p>\\1</p></pre>", $text);
  $text = preg_replace("/<p>\[spoiler=(.*?)\](.*?)\[\/spoiler\]<\/p>/si",
  "<aside><a href='#'><span></span>\\1</a><div><p>\\2</p></div></aside>",
  $text);
  $text = preg_replace("/<p>\[lightbox=(.*?)\](.*?)\[\/lightbox\]" .
  "<\/p>/si", "<figure class='lightbox'><a href='#'>".
  "<span>Щелкните, чтобы увеличить</span><br><img src='\\1'>" .
  "<img class='full-image' src='\\2'></a></figure>", $text);
  $text = str_replace("<p></p>", "", $text);
  return $text;
}
```

Сначала следует преобразовать все недопустимые символы HTML в соответствующие им литералы. Для этого мы применяем функцию PHP `htmlentities`. В качестве аргумента она принимает строку, а в качестве результата возвращает ту же строку, в которой все недопустимые символы уже преобразованы.

Далее мы помещаем отдельные строки содержания статьи в теги абзацев (`<p>`). Строки разделяются последовательностями символов возврата каретки и перевода строки (`"\r\n"`), и этот факт мы можем использовать для выполнения такого преобразования. Посмотрим выражение, которое делает это:

```
$text = preg_replace("/\n(.*?)\r/si", "<p>\\1</p>", "\n". $text . "\r");
```

Здесь используется функция PHP `preg_replace`. Она осуществляет замену в заданной строке всех подстрок, обозначенных регулярным выражением, другой подстрокой:

```
preg_replace(<искомая подстрока>, <заменяющая подстрока>,
<строка, в которой выполняется замена>)
```

Все аргументы задаются в виде строк. *Искомая подстрока*, как уже говорилось, указывается в виде регулярного выражения. В *заменяющей подстроке* можно использовать фрагменты, совпадающие с заданными в *искомой подстроке* группами регулярных выражений. Обработанная строка возвращается в качестве результата.

Приведенное ранее выражение заменяет последовательности вида `"\n<строка>\r"` на последовательности вида `"<p><строка></p>"`. То есть фактически выполняет преобразование отдельных строк содержания статьи в абзацы HTML. (Чтобы преобразование выполнялось корректно, в начало исходного содержания статьи мы предварительно добавляем символ перевода строки, а в конец — символ возврата каретки.)

Отметим, что в группе регулярного выражения мы поставили литерал `?`, указывающий выбирать минимальное количество символов, совпадающих с этой группой. Если мы не поставим этот литерал, то, вполне возможно, PHP включит в группу все символы, находящиеся между самым первым символом перевода строки и самым последним символом возврата каретки, что нам совсем не нужно.

Далее мы выполняем собственно преобразование тегов BBCode в соответствующие им HTML-теги и их комбинации. Рассмотрим несколько строк кода, приведенного в листинге 41.12, в качестве примеров.

- Здесь мы ищем фрагменты, заключенные в парные BBCode-теги `[b]`, и заменяем их подстроками, где те же фрагменты заключены в HTML-теги `<strong>`:

  ```
  $text = preg_replace("/\[b\](.*?)\[\/b\]/si", "<strong>\\1</strong>",
  $text);
  ```

- В парный BBCode-тег `[h1]`, задающий заголовок первого уровня, заключаются целые строки. Тогда после разбиения текста на абзацы, выполненного нами ранее, перед открывающими тегами `[h1]` будут находиться открывающие теги `<p>`, а после закрывающих тегов `[/h1]` — закрывающие теги `</p>`. Этот момент мы учли, выполняя поиск тега `[h1]`, заключенного в тег `<p>`, и заменяя все это HTML-тегом `<h2>`:

  ```
  $text = preg_replace("/<p>\[h1\](.*?)\[\/h1\]<\/p>/si",
  "<h2>\\1</h2>", $text);
  ```

- После выполнения этого выражения содержимое парного HTML-тега `<pre>` (текст фиксированного формата) у нас будет представлять набор тегов `<p>` (абзацев). Стандартом HTML такое допускается:

  ```
  $text = preg_replace("/<p>\[code\](.*?)\[\/code\]<\/p>/si",
  "<pre><p>\\1</p></pre>", $text);
  ```

Результирующий HTML-код, скорее всего, будет содержать пустые абзацы (<p></p>), которые следует удалить. Для этого применяется выражение:

```
$text = str_replace("<p></p>", "", $text);
```

Здесь мы использовали функцию `str_replace`. Она выполняет ту же задачу, что и рассмотренная ранее функция `preg_replace`, но в качестве первого аргумента принимает обычную строку, а не регулярное выражение, вследствие чего работает быстрее. С помощью этой функции мы заменяем последовательности "<p></p>" на пустые строки, то есть фактически удаляем их.

Осталось добавить в таблицу стилей basic.css еще несколько стилей, которые зададут оформление для строк, выровненных по середине и правому краю, и тексту фиксированного форматирования:

```
.center {             text-align: center; }
.right {              text-align: right; }
pre p {               margin: 2px auto; }
pre p:first-child {  margin-top: auto; }
pre p:last-child {   margin-bottom: auto; }
```

Для всех абзацев, вложенных в тег фиксированного форматирования, кроме первого и последнего, мы задаем уменьшенные отступы сверху и снизу. После этого текст фиксированного форматирования будет выглядеть единым целым.

В завершение мы добавим в состав сайта страницу со справочником по поддерживаемым тегам BBCode. Выводить такую страницу будет действие с именем, скажем, `bbcodes` контроллера `MainController`. Сделайте это самостоятельно.

Собственно вывод статьи

Перед тем как заняться собственно выводом выбранной посетителем статьи, нам следует позаботиться о том, чтобы реализовать возврат на «правильную» страницу. Как мы помним, на страницу просмотра статьи посетитель может попасть с трех разных страниц.

Давайте определять страницу, с которой посетитель перешел на страницу просмотра статьи, по GET-параметрам, присутствующим в составе интернет-адреса:

- если присутствует параметр `search` — посетитель выполнил переход со страницы результатов поиска;
- если параметра `search` нет, но имеет место параметр `page`, — был выполнен переход со страницы списка статей, относящихся к выбранной подкатегории;
- если же отсутствуют оба этих параметра — посетитель «пришел» со страницы выбранной категории.

Определив таким образом страницу, мы сможем сформировать ее интернет-адрес, который потом, в шаблоне, подставим в гиперссылку возврата. Формирование интернет-адреса мы будем выполнять в особом методе контроллера `ArticleController`, так как эти данные впоследствии понадобятся нам в других методах-действиях того же контроллера.

Откроем файл app\Http\Controllers\ArticleController.php, где хранится код класса контроллера `ArticleController`. Добавим в него код, объявляющий этот метод (листинг 41.14), который мы назовем `getRedirect` и сделаем частным.

Листинг 41.14

```
private function getRedirect($article) {
  $params = [];
  if (request()->has("search")) {
    $params["search"] = request()->input("search");
    $params["page"] = request()->input("page");
    $c = "ArticleController@search";
  } else {
    $params["category"] = $article->subcat->cat->slug;
    if (request()->has("page")) {
      $params["subcategory"] = $article->subcat->slug;
      $params["page"] = request()->input("page");
      $c = "ArticleController@index";
    } else {
        $c = "CategoryController@view";
    }
  }
  return action($c, $params);
}
```

Как работает этот метод, в принципе, понятно. Так что не будем останавливаться на нем и продолжим работу над контроллером `ArticleController`.

За вывод содержания выбранной посетителем статьи будет «отвечать» действие `view`. Его код чрезвычайно прост:

```
public function view(Category $category, Subcategory $subcategory,
Article $article) {
  return view("articles.view", ["article" => $article,
  "h" => $this->getRedirect($article)]);
}
```

Здесь мы передаем шаблону в составе контекста данных, помимо самой отображаемой статьи, еще и интернет-адрес для гиперссылки возврата, возвращенный только что написанным нами методом `getRedirect`.

Теперь займемся шаблоном resources\views\articles\view.blade.php, который выведет страницу статьи. Его код показан в листинге 41.15.

Листинг 41.15

```
@extends(request()->route()->getController()->parentView)

@section("title", $article->title . "  ::  " . $article->subcat->name .
" ::  " . $article->subcat->cat->name)
```

```
@section("main")
  @if (request()->route()->getController()->isMobile)
    <?php
      $l = "<p><a href='/'>Главная</a>";
      if (request()->has("search")) {
        $l .= " &lt;&lt; <a href='" . action('ArticleController@search',
        ['search' => request()->input('search'),
        'page' => request()->input('page')]) . "'>Поиск</a>";
      } else {
        $l .= " &lt;&lt; <a href='" . action('CategoryController@view',
        ['category' => $article->subcat->cat->slug]) . "'>" .
        $article->subcat->cat->name . "</a>" .
        " &lt;&lt; <a href='" . action('ArticleController@index',
        ['category' => $article->subcat->cat->slug,
        'subcategory' => $article->subcat->slug,
        'page' => request()->input('page')]) . "'>" .
        $article->subcat->name . "</a>";
      }
      $l .= "</p>"
    ?>
    {!! $l !!}
  @else
    <p><a href="{{ $h }}">Список статей</a></p>
  @endif
  <h1>{{ $article->title }}</h1>
  <p class="author-created">{{ $article->written_by->name }} ||
  {{ date_format($article->created_at, "j.m.Y G:i") }}</p>
  {!! $article->formatted_content !!}
  @if (request()->route()->getController()->isMobile)
    {!! $l !!}
  @else
    <p><a href="{{ $h }}">Список статей</a></p>
  @endif
@endsection
```

Если обращение выполняется с мобильного устройства, мы формируем и сохраняем в особой переменной HTML-код, создающий набор из следующих гиперссылок:

- на главную страницу;
- на страницу поиска — если переход был выполнен со страницы поиска;
- на страницы категории и подкатегории — если переход был выполнен со страницы списка статей, относящихся к выбранной подкатегории.

Этот код мы используем для создания двух наборов гиперссылок возврата — вверху и внизу страницы — чтобы не формировать его каждый раз заново.

Если же обращение было выполнено с традиционного компьютера, мы выводим аналогичные гиперссылки, пользуясь интернет-адресом, который был сформирован методом `getRedirect` контроллера и передан шаблону в составе контекста данных.

Также обратим внимание, что для получения содержания статьи с преобразованными BBCode-тегами мы обращаемся к вычисляемому свойству `formatted_content` модели `Article`. Значение этого свойства мы выводим на экран с помощью команды Blade `{!! !!}` (подробности — в *главе 31*), чтобы содержащиеся в нем HTML-теги не подвергались предварительной обработке и были преобразованы в соответствующие элементы страницы. (Если мы применим привычную команду `{{ }}`, HTML-код будет выведен на страницу как есть.)

Внизу страницы мы выводим еще один набор гиперссылок для возврата на предыдущие страницы.

Инструменты для работы над статьями

Ранее мы дали возможность пользователю создавать, править и удалять статьи как на странице списка статей, относящихся к выбранной подкатегории, так и на странице поиска. Следовательно, нам и здесь придется реализовывать возврат на «правильную» страницу.

Давайте рассмотрим код метода-действия `input` контроллера `ArticleController`, которое выводит страницу создания или правки статьи. Он показан в листинге 41.16.

Листинг 41.16

```
public function input(Category $category, Subcategory $subcategory,
Article $article) {
  if ($article->id) {
    $this->authorize("manipulate", $article);
  } else {
    $article->subcategory = $subcategory->id;
  }
  $subcats = Subcategory::select("subcategories.id",
  "categories.name as cat", "subcategories.name as subcat")
  ->join("categories", "subcategories.category", "categories.id")
  ->orderBy("categories.order", "desc")->orderBy("categories.name")
  ->orderBy("subcategories.order", "desc")
  ->orderBy("subcategories.name")->get();
  $params = ["page" => request()->input("page")];
  if (request()->has("search")) {
    $params["search"] = request()->input("search");
    $c = "ArticleController@search";
  } else {
    $params["category"] = $category->slug;
    $params["subcategory"] = $subcategory->slug;
    $params["page"] = request()->input("page");
    $c = "ArticleController@index";
  }
```

```
  return view("articles.input", ["article" => $article,
  "subcats" => $subcats, "params" => $params, "c" => $c]);
}
```

Сначала мы проверяем, выполняется ли создание новой статьи, или же пользователь хочет исправить уже имеющуюся статью:

- ❒ если выполняется правка статьи, мы еще раз удостоверяемся, что пользователь имеет на это право. Для чего применяем метод `authorize` контроллера, описанный в *главе 33*;
- ❒ если выполняется создание статьи, мы сразу же подставляем в ее поле `subcategory` номер текущей категории, тем самым несколько облегчая задачу автору.

Не забываем сформировать перечень подкатегорий, в который включаем номера и имена подкатегорий, а также имена категорий, к которым они относятся. Этот перечень мы используем для вывода раскрывающегося списка подкатегорий на странице создания и правки статьи.

Далее определяем, с какой страницы «пришел» пользователь — со страницы поиска или страницы подкатегории. Для чего нам достаточно выяснить, присутствует ли в интернет-адресе GET-параметр `search`. На основании этого мы подготавливаем ассоциативный массив параметров и имена контроллера и действия, на которые следует выполнить перенаправление.

Выбранную статью, массив параметров и имена контроллера и действия мы передаем шаблону resources\views\articles\input.blade.php в составе контекста данных.

Фрагмент кода этого шаблона, создающий форму, раскрывающийся список подкатегорий и гиперссылку возврата, представлен в листинге 41.17. (Код, выводящий остальные элементы управления, мы создадим сами, по аналогии с предыдущими страницами такого же рода.)

Листинг 41.17

```
@extends("layouts.pc")

@push("head")
  <script src="/js/pc.js"></script>
@endpush

@section("title", ($article->id) ? $article->title : "Добавление" .
" :: Статьи")
@section("main")
  <h1>@if ($article->id) Правка статьи {{ $article->name }}
  @else Добавление статьи @endif</h1>
  <form action="{{ action('ArticleController@save', $params) }}"
  method="POST">
    . . .
```

```
    <label>Подкатегория</label>
    <select name="subcategory">
      @foreach ($subcats as $subcat)
        <option value="{{ $subcat->id }}"
        @if (old('subcategory', $article->subcategory) == $subcat->id)
        selected @endif>{{ $subcat->cat }} -
        {{ $subcat->subcat }}</option>
      @endforeach
    </select>
    @include("common.errors", ["el" => "subcategory"])
    <input type="submit" value="Сохранить">
  </form>
  <p><a href="{{ action($c, $params) }}">Список статей</a></p>
@endsection
```

Прежде всего, в составе интернет-адреса, указывающего на действие `save` контроллера `ArticleController` (он указан в атрибуте `action` тега `<form>`), мы передаем массив параметров, полученный от контроллера (см. листинг 41.16). Это позволит действию `save` определить, с какой страницы был выполнен переход на страницу правки статьи, и после сохранения статьи произвести перенаправление на нее.

Пункты раскрывающегося списка подкатегорий будут иметь надписи вида ***<категория> - <подкатегория>***. Так авторам статей будет удобнее.

Внизу страницы мы, как обычно, ставим гиперссылку возврата. Ее интернет-адрес будет сформирован с применением, опять же, полученных от контроллера параметров и имен контроллера и действия, на которые следует выполнить переход.

На этой странице у нас будет присутствовать область редактирования для написания содержания статьи. По умолчанию она будет иметь высоту в пару строк, что явно недостаточно для работы с большим текстом. Поэтому давайте растянем ее по вертикали на все пространство окна Web-обозревателя.

В шаблоне resources\views\articles\input.blade.php найдем формирующий эту область редактирования тег и привяжем к нему стилевой класс `article-content`. После чего откроем таблицу стилей pc.css и добавим в нее стиль:

```
textarea.article-content { height: calc(100vh - 110px); }
```

Здесь мы задаем для этой области редактирования высоту, равную высоте клиентской области окна Web-обозревателя за вычетом суммарной высоты «шапки», «поддона», установленных для них внутренних и внешних отступов сверху и снизу и толщины верхней и нижней сторон рамки, заданной для элементов управления.

Займемся методом-действием `save` контроллера `ArticleController`. Его код, показанный в листинге 41.18, также примечателен.

Листинг 41.18

```
. . .
use App\Http\Requests\ArticleRequest;

class ArticleController extends Controller {
  . . .
  public function save(ArticleRequest $request) {
    if ($request->has("id")) {
      $article = Article::findOrFail($request->id);
      $this->authorize("manipulate", $article);
      $article->fill($request->all())->save();
      $s = " исправлена";
    } else {
      $article = new Article($request->all());
      $article->author = auth()->id();
      if (!($article->description)) {
        $article->description = substr($article->content, 0, 254) .
        "...";
      }
      $article->save();
      $s = " создана";
    }
    return redirect($this->getRedirect($article))
    ->with("status", "Статья " . $article->title . $s);
  }
}
```

Если выполняется сохранение уже существующей статьи, мы сначала еще раз проверяем, имеет ли текущий пользователь на это право, вызвав метод `authorize` контроллера.

Если же сохраняется вновь созданная статья, мы заносим в поле `author` (автор статьи) номер текущего пользователя. Если автор не заполнил анонс статьи, мы заносим в поле `description` (которое и хранит этот анонс) первые 254 символа содержания статьи и символ многоточия. (Получить этот символ можно, воспользовавшись утилитой Таблица символов, которая поставляется в составе Windows и описывалась в *главе 2*.)

Для получения фрагмента содержания статьи мы применяем функцию `substr` PHP, которая аналогична одноименному методу объекта `String` JavaScript (см. *главу 16*) и вызывается в следующем формате:

```
substr(<строка>, <номер первого символа>[, <номер последнего символа>])
```

Если *номер последнего символа* не указан, будут извлечены все последующие символы *строки* до самого ее конца.

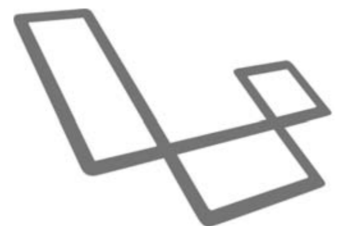
Перенаправление мы выполним по интернет-адресу, созданному методом `getRedirect` контроллера `ArticleController`.

Не забудем создать запрос формы `ArticleRequest`, в котором укажем правила валидации, и сделаем это самостоятельно: отдадим нужную команду в консоли OpenServer, откроем файл app\Http\Requests\ArticleRequest.php и исправим его код. Поля `title`, `slug` и `content` сделаем обязательными, для первых двух полей также укажем максимальную длину в 100 символов, а для поля `description` — 255 символов.

Осталось лишь создать метод-действие `destroy` класса `ArticleController`, которое будет удалять статьи. Пусть это будет вашим домашним заданием.

При написании кода этого действия нужно отметить два момента. Во-первых, в самом начале следует удостовериться, что текущий пользователь может удалять статью, вызвав все тот же метод `authorize`. Во-вторых, перенаправление следует выполнять с учетом того, что пользователь может произвести удаление статьи как на странице подкатегории, так и на странице поиска, — нужный интернет-адрес у нас генерируется методом `getRedirect`.

Закончив, выполним вход на сайт от имени любого из зарегистрированных ранее пользователей, имеющего роль администратора. Создадим несколько статей с любым содержанием, проверим работу пагинатора, операций правки, удаления и поиска, проверим также, выводятся ли пять последних добавленных статей на странице категории. Выйдем с сайта, зайдем под именем другого пользователя, теперь уже с ролью автора, добавим пару статей, проверим, действительно ли мы не можем править и удалять статьи, написанные администратором. Напоследок проверим работу «мобильной» редакции сайта.



ГЛАВА 42

Комментарии

В этой главе мы сделаем так, чтобы посетители смогли оставлять комментарии к статьям. И, разумеется, вручим редакторам и администраторам инструменты для правки и удаления комментариев.

Маршруты

Мы уже знаем, в каком файле хранятся маршруты сайта. Добавим к ним еще несколько:

```
Route::get("/comments", "CommentController@index");
Route::get("/comments/create", "CommentController@input");
Route::post("/comments", "CommentController@create");
Route::get("/comments/{comment}/edit", "CommentController@input");
Route::put("/comments", "CommentController@save");
Route::get("/comments/{comment}/delete", "CommentController@destroy");
```

Интернет-адрес **/comments** мы связываем с действием `index` контроллера `CommentController`. Это действие будет выводить страницу списка комментариев, доступную лишь редакторам и администраторам и позволяющую править и удалять комментарии.

Для создания комментариев мы используем действие `create` того же контроллера, а для правки — действие `save`.

Миграция и модель

Под хранение комментариев мы отведем таблицу `comments`. Ее структура, а точнее, все ее значимые поля, представлена в табл. 42.1.

Мы дадим возможность редакторам и администраторам скрывать отдельные комментарии, занося в поле `hidden` значение `true`. Это позволит временно исключить вывод подозрительного комментария, не удаляя его из таблицы.

Таблица 42.1. *Структура таблицы* `comments`

Имя поля	Тип поля	Параметры поля	Назначение поля
`author`	`VARCHAR`	255 символов, обязательное	Имя автора комментария
`email`	`VARCHAR`	255 символов, обязательное	Адрес электронной почты автора комментария
`content`	`TEXT`	Обязательное	Содержание комментария
`article`	`INT`	Беззнаковое, индекс	Статья, к которой был оставлен комментарий
`hidden`	`BOOLEAN`	Значение по умолчанию — `false`, индекс	Является ли комментарий скрытым

Еще мы создадим связь «один-ко-многим» между таблицами `articles` (она станет первичной) и `comments` (вторичная таблица). Поле `article` второй таблицы станет внешним ключом. Не забудем указать необходимость удаления связанных записей вторичной таблицы при удалении записи таблицы первичной — таким образом мы обеспечим удаление комментариев после удаления статьи, для которой они были оставлены.

Создадим через консоль OpenServer миграцию, откроем файл, где хранится ее код, и внесем в него необходимые правки. Исправленный код метода `up` приведен в листинге 42.1.

Листинг 42.1

```
public function up() {
  Schema::create('comments', function (Blueprint $table) {
    $table->increments('id');
    $table->string('author');
    $table->string('email');
    $table->text('content');
    $table->integer('article')->unsigned();
    $table->boolean('hidden')->default(false)->index();
    $table->foreign('article')->references('id')->on('articles')
    ->onDelete('cascade')->onUpdate('cascade');
    $table->timestamps();
  });
}
```

Выполнив через консоль OpenServer миграцию, создадим модель `Comment`. Откроем файл, где хранится ее код (где искать файлы моделей, равно как и прочих программных модулей сайта, мы уже знаем), и внесем необходимые правки (листинг 42.2).

Листинг 42.2

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model {
  protected $fillable = ["author", "email", "content", "article",
  "hidden"];

  public function art() {
    return $this->belongsTo("App\Article", "article", "id");
  }

  public function getFormattedContentAttribute() {
    $text = htmlentities($this->attributes["content"]);
    $text = preg_replace("/\n(.*?)\r/si", "<p>\\1</p>", "\n". $text .
    "\r");
    return $text;
  }}
```

Здесь мы, помимо всего прочего, объявили метод `getFormattedContentAttribute`, создающий вычисляемое поле `formatted_content` с возможностью только извлечения значения. Оно всего лишь преобразует недопустимые символы в литералы и превращает отдельные строки в абзацы, используя для этого код, аналогичный написанному нами в *главе 41* (см. листинг 41.13).

И, наконец, откроем файл с кодом модели `Article` и добавим в ее класс метод, создающий связь с моделью `Comments`:

```
public function comments() {
  return $this->hasMany("App\Comment", "article", "id");
}
```

Комментарии, относящиеся к выбранной статье

Список комментариев, относящихся к выбранной посетителем статье, мы выведем непосредственно на странице статьи. Там же мы поместим форму для добавления нового комментария.

Чтобы вывести список комментариев, относящихся к статье, нам придется исправить код метода-действия `view` контроллера `ArticleController`. Исправленный код приведен в листинге 42.3.

Листинг 42.3

```
. . .
use App\Comment;
use Illuminate\Support\Facades\Auth;

class ArticleController extends Controller {
  . . .
  public function view(Category $category, Subcategory $subcategory,
  Article $article) {
    $comments = Comment::select("author", "email", "content",
    "created_at", "hidden")
    ->where("article", $article->id)->oldest()->get();
    $comment = new Comment(["article" => $article->id]);
    if (auth()->check()) {
      $comment->author = Auth::user()->name;
      $comment->email = Auth::user()->email;
    }
    return view("articles.view", ["article" => $article,
    "h" => $this->getRedirect($article), "comments" => $comments,
    "comment" => $comment]);
  }
  . . .
}
```

Сначала мы извлекаем список комментариев, относящихся к выбранной статье, сортируя их в порядке от более старых к более новым. И сразу же после этого создаем новый комментарий, занеся в его поле `article` номер текущей статьи. Если текущий посетитель выполнил вход на сайт, мы извлекаем и подставляем в комментарий его имя и адрес электронной почты. Созданный таким образом комментарий, наряду со списком комментариев, мы передаем шаблону в составе контекста данных.

Теперь исправим шаблон, генерирующий страницу статьи (он хранится в файле resources\views\articles\view.blade.php). Мы вставим в него код, выводящий список комментариев и форму для добавления нового комментария (в ней изначально будут выводиться данные комментария, созданного действием `view`). Добавленный в шаблон код можно увидеть в листинге 42.4 — его следует поместить между содержанием статьи и нижней гиперссылкой возврата на список статей.

Листинг 42.4

```
<p> </p>
<p>Добавить комментарий:</p>
<form method="POST" action="{{ action('CommentController@create',
['search' => request()->input('search'),
'page' => request()->input('page')]) }}">
```

```
  {{ csrf_field() }}
  <input type="hidden" name="article"
  value="{{ old('article', $comment->article) }}">
  <label>Автор</label>
  <input type="text" name="author"
  value="{{ old('author', $comment->author) }}" required>
  @include("common.errors", ["el" => "author"])
  <label>E-mail автора</label>
  <input type="email" name="email"
  value="{{ old('email', $comment->email) }}" required>
  @include("common.errors", ["el" => "email"])
  <label>Содержание</label>
  <textarea name="content" class="comment-content" required>
  {{ old('content', $comment->content) }}</textarea>
  @include("common.errors", ["el" => "content"])
  <label>Введите символы с изображения</label>
  <div><img src="{{ captcha_src() }}"></div>
  <input type="text" name="captcha" required>
  @include("common.errors", ["el" => "captcha"])
  <input type="submit" value="Сохранить">
</form>
@if (count($comments) == 0)
  <p>Комментариев пока нет.</p>
@else
  <p>Комментарии (всего {{ count($comments) }})</p>
  @foreach ($comments as $com)
    @if ($com->hidden)
      <p>Комментарий скрыт.</p>
    @else
      <article class="comment" @if ($loop->last) id="last_comment"
      @endif>
        <p class="author-created">{{ $com->author }}
        ({{ $com->email }}) ||
        {{ date_format($com->created_at, "j.m.Y G:i") }}</p>
        {!! $com->formatted_content !!}
      </article>
    @endif
  @endforeach
@endif
```

В принципе, ничего особо сложного здесь нет. Нужно отметить лишь несколько примечательных моментов.

Занесенные в форму данные будут отправлены действию `create` контроллера `CommentController` (его мы напишем позже). В интернет-адрес, указывающий на это действие, мы добавим GET-параметры `search` и `page`, — эти параметры действие `create` потом «вернет» действию `view` контроллера `ArticleController`.

В состав формы добавления комментария мы поместили поле CAPTCHA — так мы обезопасимся от спамеров. Также не забудем поместить в нее скрытое поле для хранения номера статьи, с которой будет связан комментарий.

Если комментарии к выбранной статье отсутствуют, мы выводим соответствующее сообщение. Если же комментарии есть, мы выводим их количество, а уже потом — сами комментарии. На место скрытого комментария мы помещаем надпись **Комментарий скрыт**.

Тегу `<article>`, создающему последний комментарий, мы даем уникальное имя `last_comment`. Это позволит нам после добавления нового комментария прокрутить страницу таким образом, чтобы только что добавленный комментарий стал видим посетителю.

Добавим в таблицу стилей basic.css два следующих стиля:

```
article.comment {          padding: 8px;
                           margin: 20px 0px; }
textarea.comment-content { height: 10em; }
```

Первый задаст отступы для тега `<article>`, создающего комментарий, а второй — увеличит высоту области редактирования, в которую заносится содержание комментария, до десяти строк текста, чтобы посетителю было удобнее.

Займемся средствами для сохранения новых комментариев. Создадим запрос формы `CommentRequest` и укажем в нем правила валидации. Пусть поля автора, адреса электронной почты и содержания будут обязательными, для первых двух полей укажем максимальную длину в 255 символов, для поля, куда заносится адрес электронной почты, — что его значением должен быть адрес электронной почты, а для поля `hidden` — логический тип данных. Укажем сообщения об ошибках.

Важное замечание: в запросе формы `CommentRequest` не следует задавать правило для поля CAPTCHA. Дело в том, что CAPTCHA нужна только в форме создания комментария, и мы выполним проверку на правильность ее ввода непосредственно в действии `create` контроллера `CommentController`. В форме правки комментария, доступной только редакторам и администраторам, CAPTCHA излишня.

Создадим контроллер `CommentController` и добавим в него метод-действие `create`, которое и выполнит сохранение нового комментария. Его код показан в листинге 42.5.

Листинг 42.5

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

use Illuminate\Http\Request;
use App\Comment;
use App\Http\Requests\CommentRequest;

class CommentController extends Controller {
  public function __construct() {
```

```
    parent::__construct();
    $this->middleware("auth")->except("create");
  }

  public function create(CommentRequest $request) {
    $this->validate($request, ["captcha" => "captcha"],
    ["captcha.captcha" => "Введены не те символы"]);
    $comment = Comment::create($request->all());
    return redirect(action("ArticleController@view", [
    "category" => $comment->art->subcat->cat->slug,
    "subcategory" => $comment->art->subcat->slug,
    "article" => $comment->art->slug,
    "search" => request()->input("search"),
    "page" => request()->input("page")]) . "#last_comment");
  }
}
```

Этот контроллер, помимо добавления комментария, также реализует работу с комментариями — их правку и удаление. Соответствующие инструменты должны быть доступны только редакторам и администраторам, поэтому в конструкторе класса контроллера мы указали посредник `auth` для всех действий этого контроллера, кроме `create` (иначе гости не будут иметь возможности оставлять комментарии).

В действии `create` мы предварительно проверяем, правильно ли введена CAPTCHA, применив для этого знакомый нам по *главе 32* метод `validate` контроллера. Далее мы сохраняем комментарий и выполняем перенаправление по интернет-адресу, указывающему на действие `view` контроллера `ArticleController`, то есть на страницу статьи. Отметим, что мы добавляем к этому интернет-адресу имя якоря `last_comment`, в результате чего страница будет прокручена таким образом, чтобы последний добавленный комментарий (которому мы дали имя `last_comment`) стал видимым.

Напоследок попробуем добавить несколько комментариев к разным статьям и посмотрим, что из этого получится.

Страницы для работы с комментариями

Осталось создать страницы для работы с комментариями: просмотра их списка с возможностью поиска, правки и удаления выбранного комментария. Все эти операции будет выполнять контроллер `CommentController`.

Действие `index` выполнит вывод комментариев с учетом параметров поиска, заданных пользователем: искомого ключевого слова и признака, следует ли выводить только скрытые комментарии. Для указания параметров поиска мы в шаблоне создадим форму с полем ввода `search` (ключевое слово) и флажком `hdn` (выводить только скрытые комментарии). Код действия можно увидеть в листинге 42.6.

Листинг 42.6

```
public function index() {
  $comments = Comment::latest();
  if (request()->has("hdn")) {
    $comments = $comments->where("hidden", true);
  }
  if (request()->has("search")) {
    $s = request()->input("search");
    $comments = $comments->where([["author", "like", "%" . $s . "%"],
    ["content", "like", "%" . $s . "%", "or"]]);
  }
  $comments = $comments->paginate(1);
  if (request()->has("search")) {
    $comments->appends("search", $s);
  }
  if (request()->has("hdn")) {
    $comments->appends("hdn", 1);
  }
  return view("comments.index", ["comments" => $comments]);
}
```

Если в запросе присутствует GET-параметр `hdn`, то есть флажок, указывающий выводить лишь скрытые комментарии, установлен, мы отбираем только комментарии, в поле `hidden` которых хранится значение `true`. Если в запросе есть GET-параметр `search`, то есть пользователь указал ключевое слово для поиска, мы отбираем комментарии, в полях `author` (автор комментария) или `content` (содержание комментария) которых встречается заданное ключевое слово.

Далее мы создаем на основе полученного списка комментариев пагинатор с «размером» страницы равным 1 — также для целей отладки. И обязательно добавляем к основному интернет-адресу, на основе которого будут формироваться гиперссылки пагинатора, GET-параметры `search` и `hdn`, разумеется, если таковые есть в запросе.

Показанный в листинге 42.7 код шаблона resources\views\comments\index.blade.php, создающего страницу списка комментариев, довольно велик. Но ничего особо сложного в нем нет — он построен на тех же принципах, что и код шаблона, выводящего результаты поиска статей (см. листинг 41.12).

Листинг 42.7

```
@extends("layouts.pc")

@push("head")
  <script src="/js/pc.js"></script>
@endpush
```

```
@section("title", "Комментарии")
@section("main")
  <h1>Комментарии</h1>
  <form action="/comments" method="get">
    <input type="text" name="search"
    value="{{ request()->input('search') }}" class="wide">
    <label>Показывать только скрытые</label>
    <input type="checkbox" name="hdn" value="1"
    @if (request()->has('hdn')) checked @endif>
    <input type="submit" value="Найти">
  </form>
  @foreach ($comments as $comment)
    <article class="comment">
      <?php
        $params = [
          "comment" => $comment->id,
          "page" => (request()->has("page")) ?
          request()->input("page") : 1,
        ];
        if (request()->has("search")) {
          $params["search"] = request()->input("search");
        }
        if (request()->has("hdn")) {
          $params["hdn"] = 1;
        }
      ?>
      @if ($comment->hidden)
        <p class="comment-hidden">Комментарий скрыт</p>
      @endif
      <p class="author-created">{{ $comment->author }}
      ({{ $comment->email }}) ||
      {{ date_format($comment->created_at, "j.m.Y G:i") }}</p>
      {!! $comment->formatted_content !!}
      @if ($comment->hidden)
        <p class="comment-hidden">Комментарий скрыт</p>
      @endif
      @can("manipulate", $comment)
        <div class="links">
          <a href=" {{ action('CommentController@input', $params) }}">
          Исправить</a>
          <a href=" {{ action('CommentController@destroy', $params) }}"
          class="adel">Удалить</a>
        </div>
      @endcan
    </article>
  @endforeach
  {{ $comments->links() }}
@endsection
```

Непосредственно под заголовком мы помещаем форму для указания параметров поиска. Флажок `hdn` мы делаем установленным, если в запросе присутствует GET-параметр `hdn` (то есть этот флажок был установлен ранее). И не забываем задать для него в качестве значения 1.

Далее мы подготавливаем ассоциативный массив параметров, с применением которых будут сгенерированы интернет-адреса гиперссылок, указывающих на действия правки и удаления комментариев. Эти параметры включают номер текущей страницы, искомое ключевое слово и признак того, нужно ли вывести только скрытые комментарии, — они будут использованы для возврата на «правильную» страницу списка комментариев.

После этого мы выводим сам список комментариев. Отметим, что скрытые комментарии также выводятся (редакторы и администраторы должны иметь доступ ко всем комментариям, даже скрытым), но помечаются при этом надписями **Комментарий скрыт** вверху и внизу (для этих надписей следует задать подходящее представление, чтобы сделать их более заметными).

Код методов-действий `input` и `save`, а также частного метода `getParams`, показан в листинге 42.8. Здесь нам также, в принципе, все знакомо по аналогичным действиям контроллера `ArticleController` (см. *главу 41*).

Листинг 42.8

```
class CommentController extends Controller {
  private function getParams() {
    $params = ["page" => request()->input("page")];
    if (request()->has("search")) {
      $params["search"] = request()->input("search");
    }
    if (request()->has("hdn")) {
      $params["hdn"] = 1;
    }
    return $params;
  }
  . . .
  public function input(Comment $comment) {
    $this->authorize("manipulate", $comment);
    return view("comments.input", ["comment" => $comment,
    "params" => $this->getParams()]);
  }

  public function save(CommentRequest $request) {
    $comment = Comment::findOrFail($request->id);
    $this->authorize("manipulate", $comment);
    $comment->fill($request->all());
    $comment->hidden = ($request->has("hidden")) ? true : false;
    $comment->save();
```

```
    return redirect()->action("CommentController@index",
    $this->getParams())->with("status", "Комментарий исправлен");
  }
}
```

Частный метод `getParams` создает массив полученных в составе интернет-адреса GET-параметров.

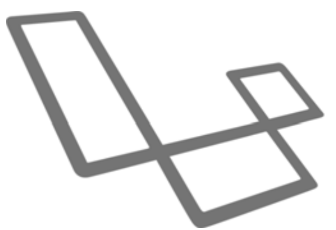
Действие `input` просто выводит страницу правки комментария и передает при этом соответствующему шаблону массив параметров, сформированный методом `getParams`, — эти параметры будут использованы для генерирования интернет-адресов в шаблоне. Вот фрагмент кода шаблона resources\views\comments\input.blade.php, в котором формируются эти интернет-адреса:

```
<form action="{{ action('CommentController@save', $params) }}"
method="POST">
  . . .
</form>
<p><a href="{{ action('CommentController@index', $params) }}">Список
комментариев</a></p>
```

Действие `save` сохраняет исправленный комментарий, — отметим, как сохраняется значение поля `hidden` (о сохранении значений логических полей говорилось в *главе 32*). Также оно выполняет перенаправление на корректную страницу списка комментариев, для чего применяет сформированный тем же методом `getParams` массив параметров.

Код действия `destroy` просто выполнит удаление выбранного комментария, а также произведет перенаправление на корректную страницу списка комментариев. Его код очень прост, и мы можем написать его самостоятельно. Также самостоятельно мы напишем шаблон resources\views\comments\input.blade.php, политику `CommentPolicy`, которая позволит править и удалять комментарии только редакторам и администраторам, и добавим в панель навигации пункт, который выведет пользователя на список комментариев.

Закончив, попробуем добавить комментарии к различным статьям. После этого выполним вход под именем пользователя, обладающего достаточными правами, и проверим, как работает правка и удаление комментариев.



ГЛАВА 43

Хранилище файлов

Ранее мы планировали позволить авторам статей вставлять в их текст изображения, аудио- и видеоролики и ссылки на архивы с исходными текстами приложений, дистрибутивные файлы самих приложений и прочие файлы, которые они собираются опубликовать вместе со статьями. Настала пора сделать это — реализовать хранение выгруженных пользователями файлов и вставку ссылок на них в текст статей.

Для каждого из выгруженных файлов мы будем сохранять описание. Оно будет выводиться в списке файлов и позволит пользователю сразу понять, что собой представляет тот или иной файл.

Мы используем для хранения выгруженных файлов локальный диск, то есть папку storage\app (о файловых хранилищах и дисках рассказывалось в *главе 32*). Понятно, непосредственно загрузить файл из этой папки не получится, и для реализации отправки сохраненного файла Web-обозревателю нам придется написать соответствующее действие в контроллере.

Маршруты

Создадим следующие маршруты:

```
Route::get("/files", "FileController@index");
Route::post("/files", "FileController@create");
Route::get("/files/{file}", "FileController@get");
Route::get("/files/{file}/delete", "FileController@destroy");
```

Действие `index` контроллера `FileController` выведет список файлов, действие `create` сохранит выгруженный файл, действие `get` — отправит файл Web-обозревателю для вывода или загрузки, а действие `destroy` — удалит.

Миграция и модель

Миграция для таблицы списка файлов будет достаточно тривиальной. А вот модель включит в свой состав два дополнительных метода… Ну, да обо всем по порядку!

Миграция

Для хранения сведений обо всех выгруженных и занесенных в хранилище файлах мы создадим в базе данных таблицу `files`. Структура ее представлена в табл. 43.1.

Таблица 43.1. *Структура таблицы* `files`

Имя поля	Тип поля	Параметры поля	Назначение поля
`type`	`TINYINT`	Беззнаковое, обязательное, индекс	Тип выгруженного файла
`description`	`VARCHAR`	Строковое, 100 символов, обязательное	Описание выгруженного файла
`extension`	`VARCHAR`	Строковое, 10 символов, обязательное	Расширение выгруженного файла
`user`	`INT`	Беззнаковое, индекс	Пользователь, выгрузивший файл

Поле `type` будет хранить тип выгруженного файла в виде целого числа:

- 0 — изображение;
- 1 — аудиофайл;
- 2 — видеофайл;
- 3 — сторонний, или прочий, файл (архив, исполняемый или файл другого типа).

В поле `extension` мы сохраним расширение файла — оно понадобится нам при формировании пути для отправки файла Web-обозревателю. Дадим этому полю размер в 10 символов — расширения у файлов сейчас встречаются довольно длинные...

И не забудем создать связь «один-ко-многим» между таблицами `users` (первичная таблица) и `files` (вторичная таблица). Поле `user` второй таблицы станет внешним ключом. Укажем необходимость удаления связанных записей вторичной таблицы при удалении записи таблицы первичной — таким образом мы обеспечим удаление файлов после удаления выгрузившего их пользователя.

Код метода `up` класса созданной нами миграции `CreateFilesTable`, которая сформирует таблицу `files`, показан в листинге 43.1.

Листинг 43.1

```
public function up() {
  Schema::create('files', function (Blueprint $table) {
    $table->increments('id');
    $table->tinyInteger('type')->unsigned()->index();
    $table->string('description', 100);
    $table->string('extension', 10);
```

```
    $table->integer('user')->unsigned();
    $table->foreign('user')->references('id')->on('users')
    ->onDelete('cascade')->onUpdate('cascade');
    $table->timestamps();
  });
}
```

Модель

Давайте поместим всю функциональность по занесению выгруженного файла в таблицу `files` и собственно сохранения файла в хранилище Laravel в класс модели. Для этого добавим в него статический метод, выполняющий сохранение файла, и обычный метод, который удалит файл.

Полный код модели `File` представлен в листинге 43.2. Давайте внимательно на него посмотрим.

Листинг 43.2

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Storage;

class File extends Model {
  protected $fillable = ["user", "type", "extension", "description"];

  public function uploaded_by() {
    return $this->belongsTo("App\User", "user", "id");
  }

  static public function saveFile($file, $type, $desc) {
    $f = File::create([
      "user" => auth()->id(),
      "type" => $type,
      "extension" => $file->getClientOriginalExtension(),
      "description" => $desc,
    ]);
    $file->storeAs("", $f->id . "." . $f->extension);
  }

  public function deleteFile() {
    Storage::delete($this->id . "." . $this->extension);
    $this->delete();
  }
}
```

Мы указали список полей, вовлеченных в массовое присваивание, и объявили метод, создающий связь с моделью `User`. Все это нам уже знакомо.

Далее мы объявили статический метод `saveFile`, который выполнит регистрацию выгруженного файла в таблице `files` и сразу же сохранит его в хранилище. В качестве параметров этот метод принимает сам файл, его тип и описание, которые и сохранит в новой записи таблицы. Далее он сохранит файл под именем вида *<номер файла>.<изначальное расширение>*, где *номер файла* будет взят из поля `id` записи. (О сохранении выгруженных файлов и работе с ними рассказывалось в *главе 32*.)

Но зачем такие сложности? Почему бы просто не сохранить файл под его изначальным именем и занести это имя в запись таблицы `files`? Дело в ошибке, которая присутствует в PHP, начиная с самых первых версий этой платформы, и из-за которой платформа преобразует все встречающиеся в именах файлов буквы, отличные от латиницы, в нечитаемые последовательности знаков. Избежать ее можно, сохранив все выгруженные файлы под некими фиктивными именами, составленными исключительно из символов латиницы и цифр, — в качестве таких имен могут выступать уникальные номера файлов, как в нашем случае.

Метод `deleteFile` удалит сразу и запись таблицы, хранящей сведения о выгруженном файле, и сам этот файл.

Не забудем добавить в класс модели `User` метод, устанавливающий связь с моделью `File`:

```
public function files() {
  return $this->hasMany("App\File", "user", "id");
}
```

Контроллер

Контроллер `FileController` у нас также будет весьма примечательным. Хотя бы потому, что его действия будут отправлять JSON-данные (чего мы еще не делали). Поэтому рассмотрим код этого контроллера по частям.

В конструкторе класса контроллера зададим посредник `auth` для всех действий, кроме `get` (оно будет выполнять отправку файла Web-обозревателю и, таким образом, должно быть доступно всем посетителям):

```
class FileController extends Controller {
  public function __construct() {
    parent::__construct();
    $this->middleware("auth")->except("get");
  }
}
```

Действие `index` сформирует список файлов, разобьет его на страницы с помощью пагинатора и отправит его Web-сценарию, выполняющему вывод этого списка на экран, в формате JSON. Здесь мы используем упрощенный пагинатор (см. *главу 32*),

поскольку выводить гиперссылки, указывающие на отдельные страницы, нам не нужно.

Действие `index` будет принимать следующие GET-параметры:

- `type` — тип файлов, которые нужно включить в список;
- `count` — количество файлов, одновременно выводящихся на экран («размер» страницы пагинатора). Эту величину будет вычислять особый Web-сценарий непосредственно после вывода на экран страницы создания и правки статьи на основе текущей ширины окна Web-обозревателя;
- и, разумеется, `page` — номер текущей страницы пагинатора.

Готовый код этого действия приведен в листинге 43.3. Он довольно прост и использует лишь приемы, знакомые нам по предыдущим главам.

Листинг 43.3

```
. . .
use App\File;
class FileController extends Controller {
  . . .
  public function index() {
    $type = (request()->has("type")) ? request()->input("type") : 0;
    $count = request()->input("count");
    $files = File::select("id", "type", "description")
    ->where("user", auth()->id())->where("type", $type)
    ->latest()->simplePaginate($count);
    $files->appends(["type" => $type, "count" => $count]);
    return $files;
  }
}
```

Здесь мы отбираем в список лишь файлы, выгруженные текущим пользователем и относящиеся к типу, который был получен с GET-параметром `type`.

Для выгрузки файлов мы создадим на странице создания и правки статьи четыре формы — по одной на каждый из заданных нами типов. Эти формы будут содержать поля ввода файлов с именами `imagefile`, `audiofile`, `videofile` и `otherfile`, служащие для указания, соответственно, графического, аудио-, видеофайла или файла другого типа. Еще эти формы будут содержать скрытое поле `type`, в котором указан целочисленный тип выгружаемого файла, и поле ввода `description`, где пользователь сможет задать для файла описание.

Действие `create` контроллера, сохраняющее выгруженный файл, должно определить по значению полученного POST-параметра `type` тип файла и, в зависимости от него, обратиться к POST-параметру `imagefile`, `audiofile`, `videofile` и `otherfile` для извлечения этого файла. Код этого действия показан в листинге 43.4.

Листинг 43.4

```
public function create(Request $request) {
  $type = $request->type;
  switch ($type) {
    case 0:
      $par = "imagefile";
      break;
    case 1:
      $par = "audiofile";
      break;
    case 2:
      $par = "videofile";
      break;
    case 3:
      $par = "otherfile";
      break;
  }
  File::saveFile($request->file($par), $type, $request->description);
  return "1";
}
```

Здесь для непосредственного сохранения файла мы используем объявленный ранее в модели `File` статический метод `saveFile`.

А теперь обратим внимание на следующий момент. Технология DOM 3 включает в свой состав программные инструменты для выгрузки файлов по технологии AJAX, но, к сожалению, не все Web-обозреватели поддерживают их. Нам придется применить традиционный способ отправки данных, когда действие, получающее и обрабатывающее выгруженный файл, генерирует страницу с результатами и отправляет ее Web-обозревателю. Однако в таком случае сгенерированная этим действием страница заменит в окне Web-обозревателя страницу создания и правки статьи. Что же нам делать?

Выходом может быть помещение на страницу создания и правки статьи невидимого фрейма (о фреймах говорилось в *главе 6*), в котором и будет выводиться результирующая страница. А отследить момент ее получения мы можем, привязав к фрейму обработчик события `load`. В таком случае нам даже не обязательно генерировать полноценную Web-страницу — мы можем отправить в качестве результата произвольные данные, например, строку `"1"` (как, собственно, мы и сделали в действии `create` — см. последнее выражение в листинге 43.4).

Что касается действия `destroy`, удаляющего файл, то после удаления оно должно сформировать обновленный список файлов и отправить его Web-сценарию. Проще всего сделать это, выполнив в конце кода этого действия перенаправление на ранее созданное действие `index`. Разумеется, действие `destroy` должно для этого получить GET-параметры `type`, `count` и `page`, которые оно потом передаст действию `index`. Код этого действия показан в листинге 43.5.

Листинг 43.5

```
public function destroy(File $file) {
  $file->deleteFile();
  return redirect()->action("FileController@index",
  ["type" => request()->input("type"),
  "count" => request()->input("count"),
  "page" => request()->input("page")]);
}
```

И, наконец, действие `get`, которое отправит файлы Web-обозревателю. Его код совсем невелик и показан в листинге 43.6.

Листинг 43.6

```
public function get(File $file) {
  $path = storage_path("/app/" . $file->id . "." . $file->extension);
  if ($file->type == 3) {
    return response()->download($path, $file->description . "." .
    $file->extension);
  } else {
    return response()->file($path);
  }
}
```

Графические, аудио- и видеофайлы будут отправлены Web-обозревателю для вывода на экран, а файлы прочих типов — для загрузки. При этом для загружаемого файла будет задано имя, составленное из его описания и расширения.

Осталось лишь задать для файлов правила валидации и разграничение доступа. Пусть сайт позволяет выгружать графические и прочие файлы размером не более 1 Мбайт, аудио- и видеофайлы — размером не более 16 Мбайт. Также следует разрешить удалять файлы только тому пользователю, который их выгрузил. Создадим соответствующий запрос формы, политику и внесем необходимые правки в код контроллера `FileController` самостоятельно.

Визуальная часть

Теперь — самое сложное. Мы вставим в шаблон resources\views\articles\input.blade.php интерфейс хранилища файлов и добавим в файл сценария public\js\pc.js большое количество кода, который будет выводить списки файлов различных типов, реализует вставку их интернет-адресов в текст статьи и удаление файлов.

Но сначала найдем графический файл с изображением, которое представит в списке файл прочего типа. (Мы ведь не можем показать в Web-обозревателе содержимое какого-нибудь архива или исполняемого файла, так ведь?) Сохраним этот файл в папке public, дав ему имя, скажем, archive и расширение, соответствующее формату

файла. (Автор использовал для этой цели изображение, полученное со страницы **http://www.iconsearch.ru/detailed/19418/6/**, и сохранил его под именем archive.png.)

Шаблон

Интерфейс нашего хранилища файлов мы представим в виде блокнота из четырех вкладок. На каждой из них будет помещаться список файлов соответствующего типа и форма, через которую пользователь сможет выгрузить файл на сайт.

Откроем шаблон resources\views\articles\input.blade.php и поместим необходимый код между формой для ввода статьи и гиперссылкой возврата на список статей. Код, который нам следует написать, очень велик, и мы рассмотрим его по частям.

И начнем мы с кода, который сформирует на странице сам блокнот. Он представлен в листинге 43.7 и знаком нам по *главе 37*.

Листинг 43.7

```
<div class="tab-header">
  <a href="#" id="tab1_header1" class="active">Изображения</a>
  <a href="#" id="tab1_header2">Аудио</a>
  <a href="#" id="tab1_header3">Видео</a>
  <a href="#" id="tab1_header4">Прочие файлы</a>
</div>
<div class="tab-content">
  <div id="tab1_content1" class="tab tab-visible">
    . . .
  </div>
  <div id="tab1_content2" class="tab">
    . . .
  </div>
  <div id="tab1_content3" class="tab">
    . . .
  </div>
  <div id="tab1_content4" class="tab">
    . . .
  </div>
</div>
```

Рассмотрим содержимое первой вкладки блокнота — той, где находится список графических файлов и соответствующая форма (листинг 43.8).

Листинг 43.8

```
<div id="fileList_0" class="filelist"></div>
<form action="/files" method="post" enctype="multipart/form-data"
target="fileFrame_0">
  {{ csrf_field() }}
```

```
  <input type="hidden" name="type" value="0">
  <label>Изображение</label>
  <input type="file" name="imagefile" accept="image/*" required>
  <label>Описание</label>
  <input type="text" name="description" required class="wide">
  <input type="submit" value="Выгрузить">
</form>
<iframe name="fileFrame_0"></iframe>
```

Для представления списка файлов используется блок с привязанным стилевым классом `filelist`. Он находится в самом верху вкладки.

Ниже находится форма для выгрузки нового графического файла. Она отправит все занесенные в нее данные действию `create` контроллера `FileController`. Вот входящие в нее элементы управления:

- скрытое поле, хранящее обозначение типа файлов;
- поле ввода выгружаемого файла. Мы задали для этого поля ввода имя `imagefile`, как условились ранее;
- обычное поле ввода, куда заносится описание выгружаемого файла;
- кнопка отправки данных.

В самом низу вкладки присутствует фрейм, в котором будет выводиться результат, возвращенный действием `create` контроллера `FileController`.

Содержимое второй, третьей и четвертой вкладок мы можем создать самостоятельно, взяв за основу код, показанный в листинге 43.8. Только не забудем указать правильные обозначения типов файлов, имена полей ввода файла и их параметры.

Представление

Еще нам понадобится задать для интерфейса файлового хранилища соответствующее представление, добавив в таблицу стилей public\css\pc.css необходимые стили. Рассмотрим их по очереди.

- Скрываем все фреймы. Мы же не хотим, чтобы пользователь увидел на странице какие-то непонятные цифры?

  ```
  section iframe { display: none; }
  ```

- Задаем для блока, который создаст сам список файлов, высоту, отступы, рамку и гибкую верстку (см. *главу 13*). Гибкая верстка позволит нам разместить в списке файлов отдельные позиции максимально простым способом:

  ```
  .filelist { padding: 5px;
              height: 230px;
              margin-bottom: 20px;
              border: 1px solid #cccccc;
              display: flex; }
  ```

- Задаем размеры и отступы для блоков, создающих отдельные позиции списка файлов. Также укажем для них относительное позиционирование (зачем — будет объяснено позже):

```
.filelist div.item { margin: 5px;
                     width: 242px;
                     height: 220px;
                     position: relative; }
```

- Эти стили задают представление для блоков, создающих средства для перехода между страницами списка файлов. Для них мы укажем достаточно большой размер шрифта. Первый блок мы сдвинем к левому краю списка файлов, последний (второй) — к правому краю, применив для этого прием, описанный в *главе 13*. Вложенные в эти блоки гиперссылки мы превращаем в блочные элементы и растягиваем на все пространство блока — так пользователю будет проще попасть по ним:

```
.filelist div.pag {             width: 40px;
                                height: 240px;
                                margin: 5px; }
.filelist div.pag a {           display: block;
                                font-size: 36pt;
                                text-align: center;
                                padding-top: 80px;
                                height: 110px; }
.filelist div.pag:first-child { margin-right: auto; }
.filelist div.pag:last-child {  margin-left: auto; }
```

- Каждый блок — позиция списка файлов — будет содержать другой блок, в который мы и поместим выводимый файл. Для этих блоков мы также укажем гибкую верстку и расположение позиций по вертикали — так мы сможем потом поместить каждый файл точно в середине блока:

```
.filelist div.item div.content { width: 240px;
                                 height: 180px;
                                 display: flex;
                                 flex-direction: column; }
.filelist div.item div.content * { max-width: 240px;
                                   max-height: 180px;
                                   margin: auto; }
```

- Блок с описанием файла мы выведем ниже самого файла. Здесь мы указали уменьшенный размер шрифта и завершение текста многоточием при выходе его за пределы блока. Мы сделали это, чтобы исключить увеличение размеров блока при выходе содержимого за его пределы — это может нарушить дизайн:

```
.filelist div.item div.description { font-size: smaller;
                                     text-align: center;
                                     height: 20px;
                                     text-overflow: ellipsis; }
```

- Гиперссылка для вставки файла в текст статьи будет выведена ниже описания. Превратим ее в блочный элемент, чтобы не помещать в блок:

```
.filelist div.item a.file-insert { display: block;
                                   text-align: center;
                                   height: 20px; }
```

- Блок с гиперссылкой удаления файла мы превратим в свободно позиционируемый и выведем в правом верхнем углу блока — позиции списка. Ранее мы задали для последнего относительное позиционирование, так что блок с гиперссылкой удаления файла окажется там, где мы и планировали. Отметим, что для этого блока задан белый цвет фона — так он будет лучше заметен на фоне изображения:

```
.filelist div.item div.file-delete { background-color: white;
                                     font-size: larger;
                                     font-weight: bold;
                                     position: absolute;
                                     top: 0px;
                                     right: 0px; }
```

Web-сценарий

Настала пора, вероятно, самой сложной части файлового хранилища — набора Web-сценариев, которые выведут на экран список файлов, выполнят вставку файла в текст статьи и удалят файл. Мы поместим их в файл public\js\pc.js и, поскольку их код очень велик, рассмотрим по очереди.

В листинге 43.9 показан сценарий, выполняющий различные предустановки. Его мы поместим непосредственно в обработчик события `load` окна.

Листинг 43.9

```
var itemCount = 0;
for (var i = 0; i < 4; i++) {
  var oList = document.getElementById("fileList_" + i);
  if (itemCount == 0)
    itemCount = Math.floor((oList.clientWidth - 100) / 252);
  oList.setAttribute("href", "/files?type=" + i + "&count=" + itemCount);
  getFileList(oList.getAttribute("href"), false);
}
var col = document.body.getElementsByTagName("iframe");
for (var i = 0; i < col.length; i++) {
  col[i].addEventListener("load", iframeLoad, false);
}
```

Сначала мы объявляем переменную `itemCount`, в которой будет храниться количество файлов, выводимых в списке одновременно. Это количество мы вычислим позднее и будем использовать в разных местах кода.

Далее мы в цикле получаем доступ ко всем четырем блокам — спискам файлов. Если количество одновременно выводящихся файлов еще не вычислено (переменная `itemCount` хранит число 0), мы вычисляем его. Для этого получаем ширину свободного пространства внутри первого списка файлов, вычитаем из него 100 пикселов (совокупная ширина блоков с гиперссылками «листания» списка) и делим на 252 пиксела (ширина блока — позиции списка). Отбросив у получившегося результата дробную часть (этим занимается метод `floor` объекта `Math`), мы выясним количество позиций, что поместится в списке.

Далее для каждого списка файлов мы формируем интернет-адрес, который укажет на действие `index` контроллера `FileController` и передаст в его составе тип файлов (с GET-параметром `type`) и только что вычисленное количество одновременно выводимых файлов (с GET-параметром `count`). Полученный интернет-адрес мы занесем в атрибут `href` тега `<div>`, представляющего каждый список. (Язык HTML позволяет добавлять в любые теги произвольные атрибуты, и мы воспользуемся этим.)

Напоследок мы обращаемся по этому интернет-адресу и получаем первую страницу списка файлов соответствующего типа, вызвав функцию `getFileList`, которую объявим позже. Первым параметром она принимает интернет-адрес, а вторым — признак, является ли запрос синхронным (`false`) или асинхронным (`true`). В нашем случае мы выполняем синхронный запрос, поскольку Web-обозреватель не может осуществить сразу несколько асинхронных запросов одновременно. На этом цикл закончен.

Наконец, мы получаем доступ ко всем фреймам и привязываем к ним обработчик события `load`, возникающего по окончании загрузки содержимого фрейма. Этим обработчиком события станет функция `iframeLoad`, которую мы также объявим позже.

Теперь рассмотрим код функции `getFileList` (листинг 43.10). Он очень большой, т. к. эта функция выполняет множество разных задач.

Листинг 43.10

```
function getFileList(sUrl, bIsAsync) {
  oAJAX = new XMLHttpRequest();
  oAJAX.open("GET", sUrl, bIsAsync);
  oAJAX.onreadystatechange = function() {
    if ((oAJAX.readyState == 4) && (oAJAX.status == 200)) {
      var oData = JSON.parse(oAJAX.responseText);
      if (oData.data.length > 0) {
        var oList = document.getElementById("fileList_" +
        oData.data[0].type);
        var s = "<div class='pag'>";
        if (oData.next_page_url)
          s += "<a href='" + oData.next_page_url + "'>&lt;</a>";
        s += "</div>";
        var d;
```

```
        for (var i = oData.data.length - 1; i >= 0; i--) {
          d = oData.data[i];
          s += "<div class='item'><div class='content'>";
          switch (d.type) {
            case 0:
              s += "<img src='/files/" + d.id + "'>";
              break;
            case 1:
              s += "<audio controls preload='metadata' src='/files/" +
              d.id + "'>";
              break;
            case 2:
              s += "<video controls preload='metadata' src='/files/" +
              d.id + "'>";
              break;
            case 3:
              s += "<img src='/public/archive.png'>";
              break;
          }
          s += "</div><div class='description'>" + d.description +
          "</div><a href='/files/" + d.id + "' class='file-insert' " +
          "type='" + d.type + "' description='" + d.description + "'" +
          ">Вставить</a><div class='file-delete'><a href='/files/" +
          d.id + "/delete?type=" + d.type + "&count=" + itemCount +
          "&page=" + oData.current_page + "'>X</a></div></div>";
        }
        s += "<div class='pag'>";
        if (oData.prev_page_url)
          s += "<a href='" + oData.prev_page_url + "'>&gt;</a>";
        s += "</div>";
        oList.innerHTML = s;
        var col = oList.querySelectorAll("div.pag a");
        for (var i = 0; i< col.length; i++) {
          col[i].addEventListener("click", filePagClick);
        }
        var col = oList.querySelectorAll("div.file-delete a");
        for (var i = 0; i< col.length; i++) {
          col[i].addEventListener("click", fileDeleteClick);
        }
        var col = oList.querySelectorAll("a.file-insert");
        for (var i = 0; i< col.length; i++) {
          col[i].addEventListener("click", fileInsertClick);
        }
      }
    }
  }
  oAJAX.send();
}
```

Код, отправляющий запрос и проверяющий состояние получения результата, мы рассматривать не будем — он знаком нам еще по *главе 22*. Сосредоточимся на коде, который выводит список файлов.

Сначала мы получаем доступ к списку файлов типа, чье обозначение мы приняли в составе данных, сгенерированных действием `index` контроллера `FileController`. После чего начинаем формировать элементы, которые будут выведены в этом списке.

- Блок для гиперссылки, ведущей на предыдущую страницу списка файлов. Если в составе полученных данных присутствует ее интернет-адрес, мы выводим эту гиперссылку, в противном случае блок останется пустым.
- Блок — позицию списка файлов, включающий следующие элементы:
  - блок с тегом, соответствующим файлу того или иного типа;
  - блок описания;
  - гиперссылку для вставки файла в текст статьи. В атрибуте `href` ее тега мы сохраняем интернет-адрес файла, в атрибутах тега `type` и `description` — соответственно обозначение типа и описание;
  - гиперссылку для удаления файла. Для нее мы задаем интернет-адрес, указывающий на действие `delete` контроллера `FileController` и включающий GET-параметры `type`, `count` и `page`.
- Блок для гиперссылки, ведущей на следующую страницу списка файлов. Если мы получили ее интернет-адрес, то выводим эту гиперссылку, в противном случае блок останется пустым.

Отметим, что элементы массива, представляющего список файлов, мы перебираем в направлении от конца к началу. Это нужно для того, чтобы вывести файлы в списке в порядке от выгруженных раньше к выгруженным позже.

После чего мы выводим все сгенерированные элементы в блоке списка файлов и выполняем привязку функций — обработчиков события `click`:

- к гиперссылкам «листания» списка привязываем функцию `filePagClick`;
- к гиперссылкам удаления файлов — функцию `fileDeleteClick`;
- к гиперссылкам вставки файлов в текст статьи — функцию `fileInsertClick`.

Последняя для нас наиболее интересна. Давайте посмотрим на ее код (листинг 43.11).

Листинг 43.11

```
function fileInsertClick(evt) {
  var oContent = document.querySelector("textarea[name=content]");
  var content = oContent.value;
  var pos = oContent.selectionStart;
  var s = content.substring(0, pos);
```

```
  var desc = evt.target.getAttribute("description");
  switch (evt.target.getAttribute("type")) {
    case "0":
      s += evt.target.getAttribute("href");
      break;
    case "1":
      s += "\r\n[audio]" + evt.target.getAttribute("href") +
      "[/audio]\r\n [sign]" + desc + "[/sign]\r\n";
      break;
    case "2":
      s += "\r\n[video]" + evt.target.getAttribute("href") +
      "[/video]\r\n [sign]" + desc + "[/sign]\r\n";
      break;
    case "3":
      s += "[url=" + evt.target.getAttribute("href") + "]" + desc +
      "[/url]";
      break;
  }
  s += content.substring(pos);
  oContent.value = s;
  evt.preventDefault();
}
```

Здесь мы получаем доступ к области редактирования, где вводится текст статьи, извлекаем содержимое этой области редактирования и текущую позицию текстового курсора в ней (через свойство `selectionStart`, рассмотренное нами в *главе 19*). Далее мы выделяем часть текста статьи, предшествующую позиции текстового курсора. (Метод `substring` объекта `String` аналогичен знакомому нам методу `substr`, но в качестве второго аргумента принимает номер конечного символа выделяемой подстроки. Если же второй аргумент не указан, выбирается вся оставшаяся часть строки.) И извлекаем из атрибута `description` тега гиперссылки, на которой щелкнули мышью, описание файла.

После этого мы получаем из других атрибутов этого же тега данные, необходимые для формирования тега BBCode, который поместит в текст страницы соответствующий файл. И формируем этот тег, добавив его к извлеченной ранее предшествующей части текста.

- Для изображения мы просто выводим указывающий на него интернет-адрес (отметим, что он фактически указывает на действие `get` контроллера `FileController`, которое отправляет файл Web-обозревателю). Это предусмотрено на тот случай, если пользователь хочет вставить изображение в тег `[lightbox]` в качестве основного изображения, выводимого в лайтбоксе, или же миниатюры.
- Аудио- и видеофайлы мы вставляем в теги `[audio]` и `[video]` соответственно и помещаем эти теги в отдельной строке, добавляя перед ними и после них симво-

лы возврата каретки и перевода строки (`"\r\n"`). Описание мы вставляем ниже этих тегов, заключив его в тег `[sign]`.

- Для прочего файла мы создаем тег гиперссылки `[url]`, указав в нем интернет-адрес этого файла. Описание файла мы помещаем внутрь тега `[url]`.

Наконец, добавляем оставшуюся часть текста статьи — ту, что находилась после текстового курсора. После чего нам останется лишь поместить получившийся текст в область редактирования и отменить поведение гиперссылки по умолчанию.

Функция `filePagClick`, обрабатывающая щелчки на гиперссылках «листания» списка и удаления файлов, очень проста:

```
function filePagClick(evt) {
  getFileList(evt.target.href, true);
  evt.preventDefault();
}
```

Она всего лишь выполняет получение страницы списка, номер которой был указан в составе интернет-адреса гиперссылки, посредством асинхронного AJAX-запроса (вторым аргументом функции `getFileList` передается значение `true`).

Функция `fileDeleteClick`, выполняющая удаление файла, немногим сложнее:

```
function fileDeleteClick(evt) {
  if (window.confirm("Удалить этот файл?"))
    getFileList(evt.target.href, true);
  evt.preventDefault();
}
```

Здесь мы сначала выводим окно-предупреждение, спрашивающее у пользователя, действительно ли он хочет удалить этот файл, и при получении положительного ответа выполняем его удаление. Для чего мы вызываем функцию `getFileList`, передавая ей первым аргументом интернет-адрес гиперссылки удаления файла, на которой пользователь щелкнул мышью. Поскольку действие `destroy` контроллера `FileController` после удаления файла производит перенаправление на действие `index`, функция `getFileList` в результате получит корректно сформированный массив JSON-данных, представляющий страницу обновленного списка файлов, и успешно обработает его.

Функция `iframeLoad`, выполняющаяся при обновлении содержимого фрейма, т. е. по окончании выгрузки файла, также не отличается сложностью:

```
function iframeLoad(evt) {
  var parent = evt.target.parentElement;
  var o = parent.querySelector("div.filelist");
  getFileList(o.getAttribute("href"), true);
  o = parent.querySelector("form");
  o.reset();
}
```

В ней мы получаем фрейм, в котором возникло событие, и добираемся до его родителя — блока, формирующего вкладку блокнота. Из родителя мы получаем доступ

к блоку — списку файлов, извлекаем из атрибута `href` его тега интернет-адрес для получения первой страницы списка и получаем эту страницу. (Поскольку все вновь выгруженные файлы добавляются, учитывая заданный нами порядок сортировки, в начало списка, после выгрузки очередного файла мы увидим его на самой первой странице.) После чего добираемся до формы, посредством которой был выгружен файл, и выполняем ее сброс.

Оценить интерфейс созданного нами файлового хранилища можно на рис. 43.1. Там представлена вкладка со списком изображений и соответствующей формой.

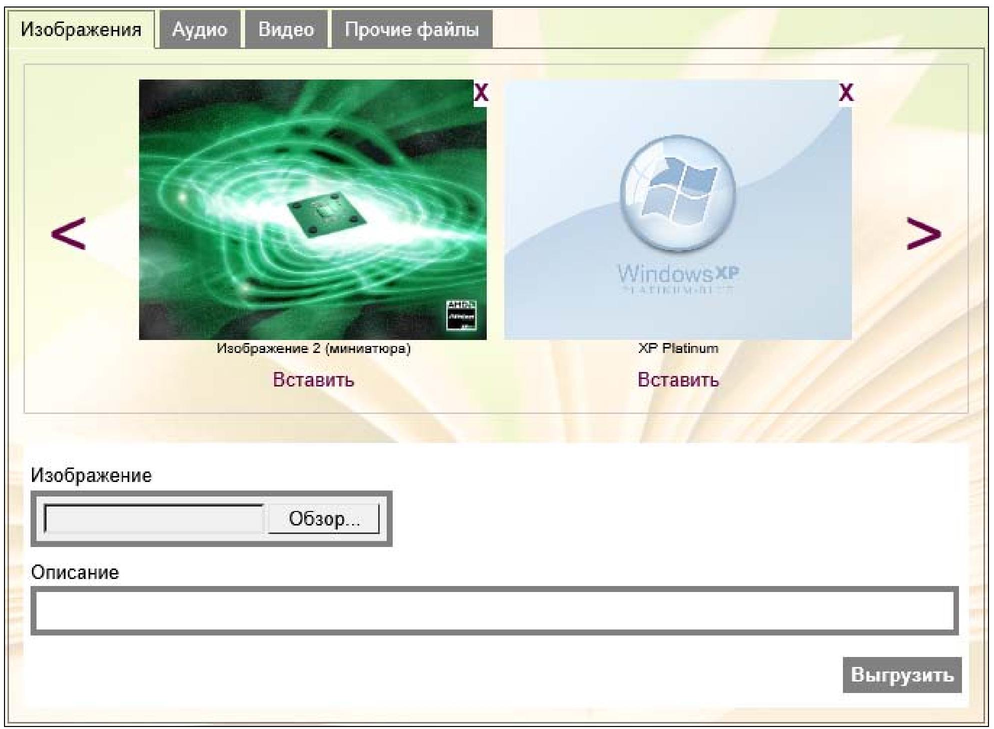
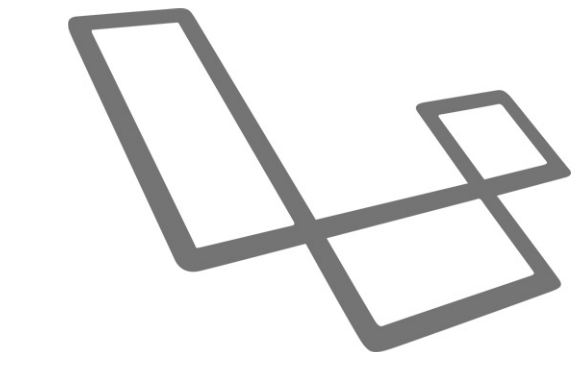


Рис. 43.1. Интерфейс файлового хранилища

Выполним вход на сайт под именем какого-либо пользователя, добавим в хранилище несколько файлов разных типов и попробуем вставить их в текст статьи. После этого войдем на сайт под именем другого пользователя и произведем те же самые действия.

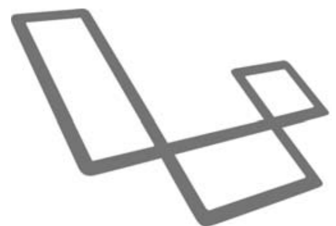
На этом разработка сайта электронных публикаций «СЭД» закончена.



III. РАЗДЕЛ 7

Нанесение последних штрихов и публикация Web-сайта

Глава 44.	Программируемая графика HTML 5
Глава 45.	Хранение данных на стороне клиента
Глава 46.	Публикация Web-сайта



ГЛАВА 44

Программируемая графика HTML 5

В этом разделе мы рассмотрим пару технологий, которые могут оказаться полезными Web-программисту, — они позволят нам расширить функциональность сайта. И начнем мы с программируемой графики HTML 5 — технологии, позволяющей нам рисовать прямо на странице произвольные изображения.

Мы также узнаем, как опубликовать готовый сайт, разработанный с применением Laravel, в Интернете — так сказать, дать ему путевку в жизнь.

Канва

Стандарт HTML 5 предусматривает для вывода программируемой графики особый элемент страницы, называемый *канвой*. В прочих элементах рисование графики не поддерживается.

Канва создается с помощью парного тега `<canvas>`. Содержимое у этого тега не указывается. Необязательные атрибуты тега `width` и `height` задают, соответственно, ширину и высоту канвы в пикселах, — если они не указаны, размеры канвы составят 300×150 пикселов.

В следующем примере мы создаем канву размерами 400×300 пикселов:

```
<canvas id="cnv" width="400" height="300"></canvas>
```

Контекст рисования

Создав на странице саму канву, мы получим ее *контекст рисования*, который можно рассматривать как набор инструментов, используемый для рисования на ней. Контекст рисования представляется экземпляром объекта `CanvasRenderingContext2D`.

Контекст рисования возвращается методом `getContext` канвы. В качестве значения единственного аргумента этого метода указывается строка `"2d"`.

В следующем примере мы получаем контекст рисования созданной ранее канвы `cnv`:

```
var oCanvas = document.getElementById("cnv");;
var ctxCanvas = oCanvas.getContext("2d");
```

Все операции по рисованию выполняются с применением методов контекста рисования. Рассмотрению таких методов и будет посвящена эта глава.

Рисование прямоугольников

Начнем мы с самых простых операций — рисования различных прямоугольников с заливкой и без нее.

Для рисования прямоугольника без заливки (т. е. одного лишь контура прямоугольника) предназначен метод `strokeRect` объекта `CanvasRenderingContext2D`:

```
<контекст рисования>.strokeRect(<горизонтальная координата>,
<вертикальная координата>, <ширина>, <высота>)
```

Все аргументы указываются в виде чисел в пикселах. Начало координат канвы находится в ее левом верхнем углу, горизонтальная координата отсчитывается по направлению вправо, а вертикальная — вниз. Результата этот метод не возвращает. Пример:

```
ctxCanvas.strokeRect(20, 20, 360, 260);
```

Метод `fillRect` рисует прямоугольник с заливкой. Вызывается он точно так же, как метод `strokeRect`. Пример:

```
ctxCanvas.fillRect(40, 40, 320, 220);
```

Метод `clearRect` очищает заданную прямоугольную область от любой присутствующей там графики. Вызывается он так же, как методы `strokeRect` и `fillRect`.

Вот пара примеров:

- рисуем большой прямоугольник с заливкой, занимающий всю канву, после чего создаем в его середине прямоугольную «прореху»:

  ```
  ctxCanvas.fillRect(0, 0, 400, 300);
  ctxCanvas.clearRect(100, 100, 200, 100);
  ```

- очищаем канву от всей присутствующей на ней графики:

  ```
  ctxCanvas.clearRect(0, 0, 400, 300);
  ```

Задание цвета, уровня прозрачности и толщины линий

Для задания цвета контура мы используем свойство `strokeStyle`. Все фигуры, которые мы впоследствии нарисуем, будут иметь контур указанного цвета. Сам цвет может быть указан любым способом, поддерживаемым CSS 3 (см. *главу 8*).

Вот пара примеров:

- все эти выражения задают для контура красный непрозрачный цвет:

  ```
  ctxCanvas.strokeStyle = "red";
  ctxCanvas.strokeStyle = "#ff0000";
  ctxCanvas.strokeStyle = "rgb(255, 0, 0)";
  ctxCanvas.strokeStyle = "rgba(255, 0, 0, 1)";
  ```

- задаем для контура темно-зеленый полупрозрачный цвет:

  ```
  ctxCanvas.strokeStyle = "rgba(0, 127, 0, 0.5)";
  ```

Изначально, сразу после загрузки и вывода канвы на страницу, линии контура будут иметь черный цвет.

Свойство `fillStyle` задает цвет заливки, который может быть задан любым поддерживаемым CSS 3 способом. По умолчанию цвет заливок также черный.

В следующем примере мы задаем для заливки синий цвет:

```
ctxCanvas.fillStyle = "rgb(0, 0, 255)";
```

Код из листинга 44.1 рисует прямоугольник, используя и для контура, и для заливки непрозрачный красный цвет, после чего поверх него рисует прямоугольник с заливкой, но уже полупрозрачным зеленым цветом.

Листинг 44.1

```
ctxCanvas.strokeStyle = "rgba(255, 0, 0, 1)";
ctxCanvas.fillStyle = "rgba(255, 0, 0, 1)";
ctxCanvas.fillRect(0, 100, 400, 100);
ctxCanvas.strokeStyle = "rgba(0, 255, 0, 0.5)";
ctxCanvas.fillStyle = "rgba(0, 255, 0, 0.5)";
ctxCanvas.fillRect(100, 0, 200, 300);
```

> ***Внимание!***
> Нельзя присваивать значение свойства `strokeStyle` свойству `fillStyle` и наоборот. Это вызовет ошибку выполнения сценария.

Свойство `lineWidth` указывает толщину линий контура в пикселах в виде числа.

В следующем примере мы задаем толщину линий равной 20 пикселам:

```
ctxCanvas.lineWidth = 20;
```

Свойство `globalAlpha` указывает уровень прозрачности для любой графики, которую мы впоследствии нарисуем. Уровень прозрачности также задается в виде числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный). Пример:

```
ctxCanvas.globalAlpha = 0.1;
```

Рисование сложных фигур

Канва поддерживает рисование более сложных, чем прямоугольники, фигур с контурами из множества прямых и кривых линий. Сейчас мы выясним, как это делается.

Как рисуются сложные контуры?

Рисование контура сложной фигуры начинается вызовом метода `beginPath`. Этот метод не принимает аргументов и не возвращает результата.

После окончания рисования сложного контура мы можем захотеть, чтобы Web-обозреватель его замкнул. Это реализует метод `closePath`, который также не принимает аргументов и не возвращает результата. После его вызова последняя точка контура будет соединена с самой первой, с которой началось его рисование.

Завершает рисование контура вызов метода `stroke` или `fill`. Первый метод просто завершает рисование контура, второй, помимо этого, замыкает контур, если он не замкнут, и рисует заливку получившейся фигуры. Оба этих метода не принимают аргументов и не возвращают результата.

А теперь рассмотрим методы, которые используются для рисования разнообразных линий, составляющих сложный контур.

Перо. Перемещение пера

Для рисования сложного контура используется концепция *пера* — воображаемого инструмента рисования. Перо можно перемещать в любую точку на канве. Рисование каждой линии контура начинается в точке, где в данный момент находится перо. После рисования каждой линии перо перемещается в ее конечную точку, из которой тут же можно начать рисование следующей линии контура.

Изначально, сразу после загрузки Web-страницы и вывода канвы, перо находится в точке с координатами (0; 0), т. е. в левом верхнем углу канвы. Переместить перо в другую точку канвы, где мы собираемся начать рисование контура, позволяет метод `moveTo`:

```
<контекст рисования>.moveTo(<горизонтальная координата>,
<вертикальная координата>)
```

Аргументы указываются в пикселах в виде чисел. Метод `moveTo` не возвращает результата. Пример:

```
ctxCanvas.moveTo(200, 150);
```

Это выражение перемещает перо в центр нашей канвы — в точку с координатами (200; 150).

Прямые линии

Прямые линии рисовать проще всего. Для этого используется метод `lineTo`:

<контекст рисования>.lineTo(*<горизонтальная координата>*,
<вертикальная координата>)

Аргументы задаются в виде чисел в пикселах. Метод `lineTo` не возвращает результата.

Код из листинга 44.2 рисует треугольник без заливки.

Листинг 44.2

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(380, 20);
ctxCanvas.lineTo(200, 280);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

Дуги

Дуги рисуются тоже довольно просто. Для этого используется метод `arc`:

<контекст рисования>.arc(*<горизонтальная координата>*,
<вертикальная координата>, *<радиус>*, *<начальный угол>*, *<конечный угол>*,
<против часовой стрелки?>)

Первые три аргумента указываются в пикселах, четвертый и пятый — в радианах. Отметим, что углы отсчитываются от горизонтальной оси. Если шестой аргумент имеет значение `true`, то дуга рисуется против часовой стрелки, а если `false` — по часовой стрелке. Метод `arc` не возвращает результата.

Тот факт, что углы задаются в радианах, несколько осложняет работу. Нам придется пересчитывать величины углов из градусов в радианы с помощью следующего выражения:

```
radians = (Math.PI / 180) * degrees;
```

Здесь переменная `degrees` хранит значение угла в градусах, а переменная `radians` будет хранить то же значение, но в радианах. Свойство `PI` объекта `Math` хранит значение числа π.

В следующем примере мы рисуем окружность без заливки:

```
ctxCanvas.beginPath();
ctxCanvas.arc(200, 150, 100, 0, Math.PI * 2, false);
ctxCanvas.stroke();
```

Кривые Безье

Для рисования кривых Безье с двумя контрольными точками используется метод `bezierCurveTo`:

```
<контекст рисования>.bezierCurveTo
(<горизонтальная координата первой контрольной точки>,
<вертикальная координата первой контрольной точки>,
<горизонтальная координата второй контрольной точки>,
<вертикальная координата второй контрольной точки>,
<горизонтальная координата конечной точки>,
<вертикальная координата конечной точки>)
```

Назначение аргументов этого метода понятно из их описания. Все они задаются в пикселах в виде чисел. Метод не возвращает результата. Пример:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.bezierCurveTo(120, 80, 160, 20, 100, 200);
ctxCanvas.stroke();
```

Рисование кривых Безье с одной контрольной точкой производит также не возвращающий результата метод `quadraticCurveTo`:

```
<контекст рисования>.quadraticCurveTo
(<горизонтальная координата контрольной точки>,
<вертикальная координата контрольной точки>,
<горизонтальная координата конечной точки>,
<вертикальная координата конечной точки>)
```

Пример:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.quadraticCurveTo(200, 100, 200, 200);
ctxCanvas.stroke();
```

Web-сценарий из листинга 44.3 рисует сектор окружности красного цвета.

Листинг 44.3

```
ctxCanvas.beginPath();
ctxCanvas.strokeStyle = "red";
ctxCanvas.fillStyle = "red";
ctxCanvas.moveTo(100, 100);
ctxCanvas.quadraticCurveTo(200, 100, 200, 200);
ctxCanvas.lineTo(100, 200);
ctxCanvas.lineTo(100, 100);
ctxCanvas.fill();
```

А код из листинга 44.4 рисует прямоугольник со скругленными углами.

Листинг 44.4

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(20, 0);
ctxCanvas.lineTo(180, 0);
ctxCanvas.quadraticCurveTo(200, 0, 200, 20);
ctxCanvas.lineTo(200, 80);
ctxCanvas.quadraticCurveTo(200, 100, 180, 100);
ctxCanvas.lineTo(20, 100);
ctxCanvas.quadraticCurveTo(0, 100, 0, 80);
ctxCanvas.lineTo(0, 20);
ctxCanvas.quadraticCurveTo(0, 0, 20, 0);
ctxCanvas.stroke();
```

Прямоугольники

Мы уже умеем рисовать прямоугольники, используя методы `strokeRect` и `fillRect`. Но прямоугольники, рисуемые этими методами, представляют собой независимые фигуры, не являющиеся частью какого-либо сложного контура. Если мы хотим нарисовать прямоугольник в составе сложного контура, нам придется прибегнуть к методу `rect`:

```
<контекст рисования>.rect(<горизонтальная координата>,
<вертикальная координата>, <ширина>, <высота>)
```

По окончании рисования прямоугольника перо будет установлено в точку с координатами (0; 0), т. е. в левый верхний угол канвы.

Код из листинга 44.5 рисует сложную фигуру, состоящую их трех накладывающихся друг на друга квадратов, и создает для нее заливку.

Листинг 44.5

```
ctxCanvas.beginPath();
ctxCanvas.rect(50, 50, 50, 50);
ctxCanvas.rect(75, 75, 50, 50);
ctxCanvas.rect(100, 100, 50, 50);
ctxCanvas.fill();
```

Задание стиля линий

Канва позволяет задать стиль линий, включающий в себя некоторые параметры, которые управляют формой их начальных и конечных точек и точек соединения линий друг с другом. Давайте их рассмотрим.

Свойство `lineCap` задает форму начальных и конечных точек линий. Его значение может быть одной из следующих строк:

- "butt" — начальная и конечная точки как таковые отсутствуют (поведение по умолчанию);
- "round" — начальная и конечная точки имеют вид кружков;
- "square" — начальная и конечная точки имеют вид квадратиков.

Код из листинга 44.6 рисует толстую прямую линию, начальная и конечная точки которой имеют вид кружков.

Листинг 44.6

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 10;
ctxCanvas.lineCap = "round";
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(180, 20);
ctxCanvas.stroke();
```

Свойство `lineJoin` задает форму точек соединения линий друг с другом. Его значение может быть одной из следующих строк:

- "miter" — точки соединения имеют вид острого или тупого угла (поведение по умолчанию);
- "round" — точки соединения, образующие острые углы, скругляются;
- "bevel" — острые углы, образуемые соединяющимися линиями, как бы срезаются.

Код из листинга 44.7 рисует контур треугольника толстыми линиями, причем точки соединения этих линий, образующие острые углы, будут срезаться.

Листинг 44.7

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 10;
ctxCanvas.lineJoin = "bevel";
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(380, 20);
ctxCanvas.lineTo(200, 280);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

Свойство `miterLimit` задает дистанцию, на которую могут выступать острые углы, образованные соединением линий, от точки соединения, когда для свойства `lineJoin` задано значение "miter". Углы, выступающие на бóльшую дистанцию, будут срезаны. Значение по умолчанию — 10 пикселов.

Вывод текста

Для вывода текста, представляющего собой один лишь контур без заливки, используется метод `strokeText`:

```
<контекст рисования>.strokeText(<выводимый текст>,
<горизонтальная координата>, <вертикальная координата>
[, <максимальная ширина>])
```

С первыми тремя аргументами все ясно. Четвертый, необязательный, аргумент указывает максимальное значение ширины, которую может принять выводимый на канву текст. Если выводимый текст получается шире, канва выводит его либо шрифтом с уменьшенной шириной символов (если этот шрифт поддерживает такое начертание), либо шрифтом меньшего размера. Метод `strokeText` не возвращает результата.

Пример:

```
ctxCanvas.strokeStyle = "blue";
ctxCanvas.strokeText("Всем привет!", 200, 50, 100);
```

Метод `fillText` выводит заданный текст в виде одной заливки, без контура. Вызывается он так же, как метод `strokeText`. Пример:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.fillText("Всем пока!", 50, 100);
```

Свойство `font` задает параметры шрифта, которым будет выводиться текст. Эти параметры указывают в том же формате, что и у значения атрибута стиля `font` (см. *главу 9*), в виде строки. Пример:

```
ctxCanvas.font = "italic 12pt Verdana";
```

Свойство `textAlign` устанавливает горизонтальное выравнивание выводимого текста относительно точки, в которой он будет выведен (координаты этой точки задаются вторым и третьим аргументами методов `strokeText` и `fillText`). Это свойство может принимать следующие значения:

- `"left"` — выравнивание по левому краю;
- `"right"` — выравнивание по правому краю;
- `"start"` — выравнивание по левому краю, если текст выводится в направлении слева направо, и по правому краю в противном случае (поведение по умолчанию);
- `"end"` — выравнивание по правому краю, если текст выводится в направлении слева направо, и по левому краю в противном случае;
- `"center"` — выравнивание по центру.

Пример:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.font = "italic 12pt Verdana";
```

```
ctxCanvas.textAlign = "center";
ctxCanvas.fillText("Всем пока!", 100, 100);
```

Свойство `textBaseline` позволяет задать вертикальное выравнивание выводимого текста относительно точки, в которой он будет выведен. Доступны следующие значения:

- `"top"` — выравнивание по верху прописных букв;
- `"hanging"` — выравнивание по верху строчных букв;
- `"middle"` — выравнивание по середине строчных букв;
- `"alphabetic"` — выравнивание по базовой линии букв европейских алфавитов (поведение по умолчанию);
- `"ideographic"` — выравнивание по базовой линии иероглифических символов (она находится чуть ниже базовой линии букв европейских алфавитов);
- `"bottom"` — выравнивание по низу букв.

Использование сложных цветов

Помимо однотонных цветов, HTML 5 позволяет использовать для линий и заливок градиенты и даже закрашивать их с помощью графических изображений.

Линейный градиент

Линейный градиент создают в три этапа. Первый этап — вызов метода `createLinearGradient` — собственно создание линейного градиента:

```
<контекст рисования>.createLinearGradient
(<горизонтальная координата начальной точки>,
<вертикальная координата начальной точки>,
<горизонтальная координата конечной точки>,
<вертикальная координата конечной точки>)
```

Координаты начальной и конечной точек градиента отсчитываются относительно канвы.

Метод `createLinearGradient` возвращает экземпляр объекта `CanvasGradient`, представляющий созданный нами линейный градиент. Мы используем его для указания цветов, формирующих градиент.

Второй этап — расстановка ключевых точек градиента. Здесь нам понадобится метод `addColorStop` объекта `CanvasGradient`:

```
<градиент>.addColorStop(<положение ключевой точки>, <цвет>)
```

Первый аргумент задается в виде числа от 0.0 (начало прямой) до 1.0 (конец прямой). Результата этот метод не возвращает.

Третий этап — использование готового линейного градиента. Для этого представляющий его экземпляр объекта `CanvasGradient` следует присвоить свойству `strokeStyle` или `fillStyle`. Пример:

```
var lgSample = ctxCanvas.createLinearGradient(0, 0, 100, 50);
lgSample.addColorStop(0, "black");
lgSample.addColorStop(0.4, "rgba(0, 0, 255, 0.5)");
lgSample.addColorStop(1, "#ff0000");
ctxCanvas.fillStyle = lgSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Здесь мы создаем линейный градиент, который будет «распространяться» по прямой с координатами начальной и конечной точек (0; 0) и (100; 50). После чего формируем на нем три ключевые точки:

- расположенную в начальной точке воображаемой прямой и задающую черный цвет;
- расположенную в точке, отстоящей на 40% длины воображаемой прямой от ее начальной точки, и задающую полупрозрачный синий цвет;
- расположенную в конечной точке воображаемой прямой и задающую красный цвет.

Наконец, используем созданный таким образом градиент для рисования прямоугольника.

А теперь нам нужно обсудить один очень важный вопрос. И рассмотрим мы его на примере созданного ранее градиента.

Предположим, что мы нарисовали на канве три прямоугольника и применили к ним наш линейный градиент. Первый прямоугольник нарисован в точке (0; 0) (в начале воображаемой прямой градиента — в его первой ключевой точке), второй — в точке (30; 20) (во второй ключевой точке), третий — в точке (80; 40) (в конце градиента — его третьей ключевой точке). Иначе говоря, мы «расставили» наши прямоугольники во всех ключевых точках градиента.

Как будут окрашены эти прямоугольники? Давайте посмотрим.

- Первый прямоугольник будет окрашен, в основном, в черный цвет, заданный в первой ключевой точке градиента.
- Второй прямоугольник будет окрашен, в основном, в полупрозрачный синий цвет, заданный во второй ключевой точке градиента.
- Третий прямоугольник будет окрашен, в основном, в красный цвет, заданный в третьей ключевой точке градиента.

Следовательно, созданный нами градиент не «втиснулся» в каждый из нарисованных прямоугольников целиком, а как бы зафиксировался на самой канве, а прямоугольники только «проявили» фрагменты этого градиента, соответствующие их размерам. Другими словами, градиент задается для целой канвы, а фигуры, к которым он применен, окрашиваются соответствующими его фрагментами.

Если мы захотим, чтобы какая-то фигура была окрашена градиентом полностью, придется задать для этого градиента такие координаты воображаемой прямой, чтобы он «покрыл» всю фигуру. Иначе фигура будет содержать только часть градиента.

Радиальный градиент

Радиальный градиент также создают в три этапа. Первый этап — вызов метода `createRadialGradient` — создание радиального градиента:

```
<контекст рисования>.createRadialGradient
(<горизонтальная координата центра внутренней окружности>,
<вертикальная координата центра внутренней окружности>,
<радиус внутренней окружности>,
<горизонтальная координата центра внешней окружности>,
<вертикальная координата центра внешней окружности>,
<радиус внешней окружности>)
```

Аргументы этого метода задают координаты центров и радиусы обеих окружностей, описывающих радиальный градиент. Они задаются в пикселах в виде чисел.

Метод `createRadialGradient` возвращает экземпляр объекта `CanvasGradient`, представляющий созданный нами радиальный градиент.

Второй этап — расстановка ключевых точек — выполняется с помощью уже знакомого нам метода `addColorStop` объекта `CanvasGradient`. Только здесь первый аргумент этого метода определит относительное положение создаваемой ключевой точки на промежутке между внутренней и внешней окружностями. Он задается в виде числа от 0.0 (начало промежутка, т. е. внутренняя окружность) до 1.0 (конец промежутка, т. е. внешняя окружность).

И третий этап — использование созданного градиента.

Листинг 44.8

```
var rgSample = ctxCanvas.createRadialGradient(100, 100, 10, 150, 100,
120);
rgSample.addColorStop(0, "#cccccc");
rgSample.addColorStop(0.8, "black");
rgSample.addColorStop(1, "#00ff00");
ctxCanvas.fillStyle = rgSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Код из листинга 44.8 создает радиальный градиент, «распространяющийся» от внутренней окружности, центр которой находится в точке с координатами (100; 100), а радиус равен 10 пикселам, к внешней окружности с центром в точке (150; 100) и радиусом в 120 пикселов. Далее добавляем три ключевые точки:

- расположенную в начальной точке воображаемого промежутка между окружностями (т. е. на внутренней окружности) и задающую серый цвет;
- расположенную в точке, отстоящей на 80% длины воображаемого промежутка между окружностями от его начальной точки, и задающую черный цвет;
- расположенную в конечной точке воображаемого промежутка между окружностями (т. е. на внешней окружности) и задающую зеленый цвет.

Радиальный градиент ведет себя точно так же, как линейный — фиксируется на канве и частично «проявляется» на фигурах, к которым применен.

Графический цвет

Графический цвет — это обычное графическое изображение, которым закрашиваются линии или заливки. Таким графическим изображением может быть содержимое как обычного графического файла, так и другой канвы.

Графический цвет создают в три этапа. Первый этап необходим только в том случае, если мы используем в качестве цвета содержимое графического файла. Файл нужно как-то загрузить, удобнее всего — с помощью объекта Web-обозревателя `Image`, который представляет графическое изображение, хранящееся в файле.

Сначала мы с помощью знакомого нам по *главе 16* оператора `new` создаем экземпляр объекта `Image`. Затем присваиваем свойству `src` этого объекта интернет-адрес загружаемого графического файла в виде строки. Все — теперь мы можем использовать этот экземпляр объекта `Image` для создания графического цвета.

Второй этап — собственно создание графического цвета с помощью метода `createPattern`:

```
<контекст рисования>.createPattern(<графическое изображение или канва>,
<режим повторения>)
```

Первый аргумент задает графическое изображение в виде экземпляра объекта `Image` или канву в виде представляющего ее экземпляра объекта.

Часто бывает так, что размеры заданного графического изображения меньше, чем фигуры, закрашиваемой графическим цветом. В этом случае изображение повторяется столько раз, чтобы полностью «вымостить» линию или заливку. Режим такого повторения задает второй аргумент метода `createPattern`. Его значение должно быть одной из следующих строк:

- `"repeat"` — изображение будет повторяться по горизонтали и вертикали;
- `"repeat-x"` — изображение будет повторяться только по горизонтали;
- `"repeat-y"` — изображение будет повторяться только по вертикали;
- `"no-repeat"` — изображение не будет повторяться никогда.

В последних трех случаях может получиться так, что часть закрашиваемого графическим цветом изображения может оказаться не покрытой им.

Метод `createPattern` возвращает экземпляр объекта `CanvasPattern`, представляющий созданный нами графический цвет.

Третий этап — использование готового графического цвета — выполняется так же, как для градиентов.

В следующем примере мы загружаем графический файл graphic_color.jpg, создаем на его основе повторяющийся по горизонтали и вертикали графический цвет и рисуем с его помощью прямоугольник:

```
var imgSample = new Image();
imgSample.src = "graphic_color.jpg";
var cpSample = ctxCanvas.createPattern(imgSample, "repeat");
ctxCanvas.fillStyle = cpSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Графический цвет не фиксируется на канве, а полностью применяется к рисуемой фигуре. В этом его принципиальное отличие от градиентов.

Вывод внешних изображений

Внешним по отношению к канве называется графическое изображение, хранящееся в отдельном файле, или содержимое другой канвы. Вывести такое изображение на канву проще всего методом `drawImage`, вызов которого записан в сокращенном формате:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
<горизонтальная координата>, <вертикальная координата>)
```

Первый аргумент задает графическое изображение в виде экземпляра объекта `Image` или канву в виде представляющего ее экземпляра объекта. Второй и третий аргументы определяют координаты точки канвы, где должен находиться левый верхний угол выводимого изображения, — они задаются в пикселах в виде чисел. Метод `drawImage` не возвращает результата.

В следующем примере мы загружаем изображение из файла someimage.jpg и выводим его на канву так, чтобы его левый верхний угол находился в точке (0; 0), т. е. в левом верхнем углу канвы:

```
var imgSample = new Image();
imgSample.src = "someimage.jpg";
ctxCanvas.drawImage(imgSample, 0, 0);
```

Если нам нужно при выводе внешнего изображения изменить его размеры, к нашим услугам расширенный формат метода `drawImage`:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
<горизонтальная координата>, <вертикальная координата>,
<ширина>, <высота>)
```

В следующем примере мы растягиваем выводимое изображение так, чтобы занять канву целиком:

```
ctxCanvas.drawImage(imgSample, 0, 0, 400, 300);
```

Рассмотрим теперь самый сложный случай — вырезание из внешнего изображения фрагмента и вывод его на канву с изменением размеров. Для этого применяется третий по счету формат метода `drawImage`:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
<горизонтальная координата вырезаемого фрагмента>,
```

```
<вертикальная координата вырезаемого фрагмента>,
<ширина вырезаемого фрагмента>, <высота вырезаемого фрагмента>,
<горизонтальная координата вывода>, <вертикальная координата вывода>,
<ширина вывода>, <высота вывода>)
```

Второй и третий аргументы определяют координаты левого верхнего угла фрагмента, вырезаемого из внешнего изображения. Они задаются относительно внешнего изображения в пикселах в виде чисел.

Четвертый и пятый аргументы определяют ширину и высоту вырезаемого из внешнего изображения фрагмента. Они также задаются относительно внешнего изображения в пикселах в виде чисел.

В следующем примере мы вырезаем из загруженного ранее изображения фрагмент с левым верхним углом в точке (20; 40), шириной в 40 и высотой в 20 пикселов и выводим этот фрагмент на канву, растягивая его так, чтобы занять канву целиком:

```
ctxCanvas.drawImage(imgSample, 20, 40, 40, 20, 0, 0, 400, 300);
```

Создание тени у рисуемой графики

Еще канва позволяет создавать тень у всех рисуемых фигур. Для задания ее параметров применяют четыре следующих свойства:

- `shadowOffsetX` — задает смещение тени по горизонтали относительно фигуры в виде числа в пикселах (значение по умолчанию — 0);
- `shadowOffsetY` — задает смещение тени по вертикали относительно фигуры в виде числа в пикселах (значение по умолчанию — 0);
- `shadowBlur` — задает степень размытия тени в виде числа. Чем больше это число, тем сильнее размыта тень (значение по умолчанию — 0, т. е. отсутствие размытия);
- `shadowColor` — задает цвет тени (по умолчанию — черный непрозрачный).

Пример:

```
ctxCanvas.shadowOffsetX = 2;
ctxCanvas.shadowOffsetY = -2;
ctxCanvas.shadowBlur = 4;
ctxCanvas.shadowColor = "rgba(128, 128, 128, 0.5)";
ctxCanvas.fillText("Двое: я и моя тень.", 150, 50);
```

Преобразования системы координат

Канва позволяет выполнять преобразования системы координат, которые можно уподобить знакомым нам по *главе 14* преобразованиям CSS. В результате их выполнения будет меняться система координат канвы, и графика, которую мы далее нарисуем, будет создаваться в уже измененной системе координат.

Сохранение и загрузка состояния

Первое, что нам нужно рассмотреть применительно к преобразованиям, — это сохранение и загрузка состояния канвы. Эти возможности нам очень пригодятся в дальнейшем.

При сохранении состояния канвы сохраняются:

- все заданные трансформации (будут описаны далее);
- значения свойств `globalAlpha`, `globalCompositeOperation` (будет описано далее), `fillStyle`, `lineCap`, `lineJoin`, `lineWidth`, `miterLimit` и `strokeStyle`;
- все заданные маски (будут описаны далее).

Сохранение состояния канвы выполняет метод `save`. Он не принимает аргументов и не возвращает результата.

Состояние канвы сохраняется в памяти компьютера и впоследствии может быть восстановлено. Более того, сохранять состояние канвы можно несколько раз, при этом все предыдущие состояния остаются в памяти, и их также можно восстановить.

Восстановить сохраненное ранее состояние можно вызовом метода `restore`. Он не принимает аргументов и не возвращает результата.

При вызове этого метода будет восстановлено самое последнее из сохраненных состояний канвы. При последующем его вызове будет восстановлено предпоследнее сохраненное состояние и т. д. Этой особенностью часто пользуются для создания сложной графики.

Перемещение начала координат канвы

Перемещение начала координат канвы в другую точку выполняется вызовом не возвращающего результата метода `translate`:

```
<контекст рисования>.translate(<горизонтальная координата>,
<вертикальная координата>)
```

Листинг 44.9

```
ctxCanvas.save();
ctxCanvas.translate(100, 100);
ctxCanvas.fillRect(0, 0, 50, 50);
ctxCanvas.translate(100, 100);
ctxCanvas.fillRect(0, 0, 50, 50);
ctxCanvas.restore();
ctxCanvas.fillRect(0, 0, 50, 50);
```

Код из листинга 44.9 делает следующее:

1. Сохраняет текущее состояние канвы.
2. Перемещает начало координат в точку (100; 100).

3. Рисует квадрат размерами 50×50 пикселов, левый верхний угол которого находится в точке начала координат.
4. Опять перемещает начало координат в точку (100; 100). Обратим внимание, что координаты этой точки теперь отсчитываются от нового начала системы координат, установленного предыдущим вызовом метода `translate`.
5. Опять рисует квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.
6. Восстанавливает сохраненное состояние канвы, в том числе и положение начала системы координат (это положение по умолчанию, т. е. левый верхний угол канвы).
7. Рисует третий по счету квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.

В результате мы увидим три квадрата, расположенные на воображаемой диагонали, тянущейся от левого верхнего угла канвы вниз и вправо.

> **ВНИМАНИЕ!**
>
> К сожалению, не существует простого способа вернуться к системе координат по умолчанию. Единственная возможность — сохранить состояние системы координат перед любыми трансформациями и потом восстановить его.

Поворот системы координат

Не возвращающий результата метод `rotate` поворачивает оси системы координат на произвольный угол вокруг точки начала координат:

```
<контекст рисования>.rotate(<угол поворота>)
```

Угол поворота задается в виде числа в радианах и отсчитывается от горизонтальной оси. Отметим, что поворот всегда производится по часовой стрелке.

Рассмотрим пример из листинга 44.10.

Листинг 44.10

```
ctxCanvas.translate(200, 150);
for (var i = 0; i < 3; i++) {
  ctxCanvas.rotate(Math.PI / 6);
  ctxCanvas.strokeRect(-50, -50, 100, 100);
}
```

Здесь выполняется сдвиг начала координат в центр канвы (точку (200; 150)), после чего система координат трижды поворачивается на $\pi/6$ радиан (30°), и в центре канвы рисуется квадрат без заливки. Обратим внимание, что каждый последующий поворот системы координат выполняется с учетом того, что она уже была повернута ранее.

Изменение масштаба системы координат

Не возвращающий результата метод `scale` изменяет масштаб системы координат канвы в бо́льшую или меньшую сторону:

```
<контекст рисования>.scale(<масштаб по горизонтали>,
<масштаб по вертикали>)
```

Аргументы этого метода задают масштаб для горизонтальной и вертикальной осей системы координат в виде чисел. Числа меньше 1.0 задают уменьшение масштаба, а числа больше 1.0 — увеличение. Если нужно оставить масштаб по какой-то из осей неизменным, достаточно указать значение 1.0 соответствующего параметра.

Листинг 44.11

```
ctxCanvas.save();
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.scale(3, 1);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
ctxCanvas.save();
ctxCanvas.scale(1, 3);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
ctxCanvas.save();
ctxCanvas.scale(3, 3);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
```

Код из листинга 44.11 делает следующее:

1. Сохраняет текущее состояние канвы.
2. Рисует квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.
3. Увеличивает масштаб горизонтальной координатной оси в 3 раза.
4. Рисует второй квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.
5. Восстанавливает сохраненное ранее состояние канвы и сохраняет его снова.
6. Увеличивает масштаб вертикальной координатной оси в 3 раза.
7. Рисует третий квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.
8. Восстанавливает сохраненное ранее состояние канвы и сохраняет его снова.
9. Увеличивает масштаб обеих координатных осей в 3 раза.

10. Рисует четвертый квадрат размерами 50×50 пикселов, левый верхний угол которого находится в начале координат.
11. Восстанавливает сохраненное ранее состояние канвы.

В результате мы увидим четыре прямоугольника с реальными размерами 50×50, 150×50, 50×150 и 150×150 пикселов.

Управление наложением графики

Когда мы рисуем новую фигуру поверх уже имеющейся на канве, новая фигура накладывается на старую, перекрывая ее. Это поведение канвы по умолчанию, которое мы можем изменить с помощью свойства `globalCompositeOperation`. Все поддерживаемые им значения приведены далее:

- `"source-over"` — новая фигура накладывается на старую, перекрывая ее (поведение по умолчанию);
- `"source-atop"` — отображается только та часть новой фигуры, которая накладывается на старую, остальная часть новой фигуры не выводится. Старая фигура выводится целиком и находится ниже новой;
- `"source-in"` — отображается только та часть новой фигуры, которая накладывается на старую. Остальные части новой и старой фигур не выводятся;
- `"source-out"` — отображается только та часть новой фигуры, которая не накладывается на старую. Остальные части новой фигуры и вся старая фигура не выводятся;
- `"destination-over"` — новая фигура перекрывается старой;
- `"destination-atop"` — отображается только та часть старой фигуры, которая накладывается на новую; остальная часть старой фигуры не выводится. Новая фигура выводится целиком и находится ниже старой;
- `"destination-in"` — отображается только та часть старой фигуры, на которую накладывается новая. Остальные части новой и старой фигур не выводятся;
- `"destination-out"` — отображается только та часть старой фигуры, на которую не накладывается новая. Остальные части новой фигуры и вся старая фигура не выводятся;
- `"lighter"` — цвета накладывающихся частей старой и новой фигур складываются, результирующий цвет получается более светлым, окрашиваются накладывающиеся части фигур;
- `"xor"` — отображаются только те части старой и новой фигур, которые не накладываются друг на друга;
- `"copy"` — выводится только новая фигура; все старые фигуры удаляются с канвы.

Заданный нами способ наложения графики действует только для графики, которую мы нарисуем после этого. На уже нарисованную графику он влияния не оказывает.

Пример:

```
ctxCanvas.fillStyle = "blue";
ctxCanvas.fillRect(0, 50, 400, 200);
ctxCanvas.fillStyle = "red";
ctxCanvas.globalCompositeOperation = "source-over";
ctxCanvas.fillRect(100, 0, 200, 300);
```

Этот код рисует два накладывающихся прямоугольника разных цветов и позволяет изучить поведение канвы при разных значениях свойства `globalCompositeOperation`. Изменяем значение этого свойства, перезагружаем страницу нажатием клавиши <F5> и смотрим, что получится.

Создание маски

О масках мы уже знаем из *главы 13*. В терминологии канвы так называется особая фигура, задающая своего рода «окно», сквозь которое будет видна часть графики, нарисованной на канве. Вся графика, не попадающая в это «окно», будет скрыта, при этом сама маска на канву не выводится.

Далее перечислены действия, необходимые для создания маски последовательности действий.

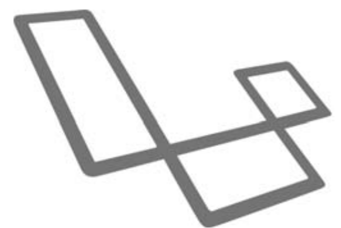
1. Рисуем сложный контур, который станет маской.
2. Обязательно делаем его закрытым.
3. Вместо вызова методов `stroke` или `fill` вызываем метод `clip`. Этот метод не принимает аргументов и не возвращает результата.
4. Рисуем графику, которая будет находиться под маской.

После этого нарисованная нами на шаге 4 графика будет частично видна сквозь маску. Требуемый результат достигнут.

Код из листинга 44.12 рисует маску в виде треугольника, а потом — прямоугольник. Часть этого прямоугольника будет видна сквозь маску.

Листинг 44.12

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 150);
ctxCanvas.lineTo(200, 0);
ctxCanvas.lineTo(200, 300);
ctxCanvas.closePath();
ctxCanvas.clip();
ctxCanvas.fillRect(0, 100, 400, 100);
```



ГЛАВА 45

Хранение данных на стороне клиента

Статьи, которые станут писать наши будущие авторы, могут иметь разный объем, в том числе и очень большой. И понятно, что за один присест достаточно большую статью не напишешь. Поэтому нам нужно предусмотреть какие-то средства для временного сохранения статей на стороне клиента.

Самый простой способ сделать это — скопировать текст статьи в Блокнот или подобный ему текстовый редактор и сохранить в файле. Но просто — не значит удобно для пользователя. Поэтому мы применим другой подход — задействуем хранилище HTML 5.

Хранилище HTML 5

Хранилище HTML 5 (не путать с файловыми хранилищами Laravel, описанными в *главе 32*) — это средство для сохранения на стороне клиента произвольных данных любого типа. Оно позволяет сохранить любое значение, указав для него уникальное имя, а впоследствии извлечь его, сославшись на заданное ранее имя, и использовать в вычислениях.

Современные Web-обозреватели предлагают разработчикам сценариев два различных хранилища.

- *Сессионное хранилище* — хранит данные, лишь пока в окне Web-обозревателя открыта текущая страница. После перехода на другую страницу или закрытия окна все сохраненные данные будут удалены. Сессионное хранилище подходит лишь для временного хранения данных в ситуации случайного обновления страницы посетителем.
- *Локальное хранилище* — сохраняет данные даже после перехода на другую страницу или закрытия окна Web-обозревателя. Позволяет хранить данные неопределенно долгое время.

Нужно помнить, что хранилище любого типа содержит данные, относящиеся к какой-то одной странице. Данные, сохраненные любой другой страницей, при этом считать невозможно — это сделано ради безопасности.

Сессионное хранилище доступно через свойство `sessionStorage`, а локальное — через свойство `localStorage`. Оба этих свойства поддерживаются объектом `Window`, представляющим окно Web-обозревателя (подробнее об этом объекте рассказывалось в *главе 20*).

Хранилище любого типа представляется объектом `Storage`, экземпляр которого, созданный самим Web-обозревателем, хранится в соответствующем свойстве объекта `Window`: `sessionStorage` или `localStorage`. Все манипуляции по сохранению и чтению данных выполняются посредством вызова методов этого объекта.

Метод `setItem` выполняет сохранение значения в хранилище:

```
<хранилище>.setItem(<имя>, <значение>)
```

Имя, под которым сохраненное значение можно будет впоследствии отыскать и прочитать, и само *значение* указываются в виде строк. Результата этот метод не возвращает.

Для чтения сохраненного ранее значения мы применим метод `getItem`:

```
<хранилище>.getItem(<имя>)
```

Имя, опять же, указывается в виде строки. Метод возвращает извлеченное из хранилища значение в виде строки, или `null`, если значение с указанным именем отсутствует.

Если понадобится удалить из хранилища сохраненное ранее значение, следует вызвать метод `removeItem`. Формат его вызова такой же, как и у метода `getItem`, а результата он не возвращает.

Единственное свойство объекта `Storage` — `length` — возвращает количество сохраненных в хранилище значений в виде числа.

Временное хранение текста статей на стороне клиента

Теоретический курс хранения данных на стороне клиента средствами HTML 5 получился совсем коротким. Настала пора практики, во время которой мы реализуем на нашем сайте временное хранение текстов статей на стороне клиента.

Сначала мы вставим в код шаблона resources\views\articles\input.blade.php фрагмент, создающий кнопки, которые будут инициировать операции сохранения статьи в локальном хранилище и извлечения ее оттуда:

```
<div id="saveLoadButtons">
  <input type="button" id="btnLoad" value="Загрузить">   
  <input type="button" id="btnSave" value="Сохранить локально">
</div>
```

Этот код можно поместить в самом начале формы для ввода статьи, перед полем для занесения ее заголовка.

Далее откроем таблицу стилей public\css\pc.css и добавим в нее два следующих стиля:

```
#saveLoadButtons {                    text-align: right; }
#saveLoadButtons input[type=button] { display: inline;
                                      float: none;
                                      clear: none; }
```

Они укажут Web-обозревателю выстроить кнопки по горизонтали, друг за другом, и сдвинуть их к правому краю родителя.

И, наконец, поместим в файл public\js\pc.js, в обработчик события `load` окна, код, выполняющий сохранение и чтение статей. Этот код приведен в листинге 45.1.

Листинг 45.1

```
var btnSave = document.getElementById("btnSave");
var btnLoad = document.getElementById("btnLoad");
if ((btnSave) && (btnLoad)) {
  var txtTitle = document.querySelector("input[name=title]");
  var txtSlug = document.querySelector("input[name=slug]");
  var txtDescription = document.querySelector↳
  ("textarea[name=description]");
  var txtContent = document.querySelector("textarea[name=content]");
  var lstSubcategory = document.querySelector↳
  ("select[name=subcategory]");
  var oLS = window.localStorage;
  btnSave.addEventListener("click", function() {
    if ((txtTitle.value) && (txtContent.value)) {
      oLS.setItem("title", txtTitle.value);
      oLS.setItem("slug", txtSlug.value);
      oLS.setItem("description", txtDescription.value);
      oLS.setItem("content", txtContent.value);
      oLS.setItem("subcategory", lstSubcategory.value);
    }
  });
  btnLoad.addEventListener("click", function() {
    if ((oLS.getItem("title")) && (oLS.getItem("content"))) {
      txtTitle.value = oLS.getItem("title");
      txtSlug.value = oLS.getItem("slug");
      txtDescription.value = oLS.getItem("description");
      txtContent.value = oLS.getItem("content");
      lstSubcategory.value = oLS.getItem("subcategory");
    }
  });
}
```

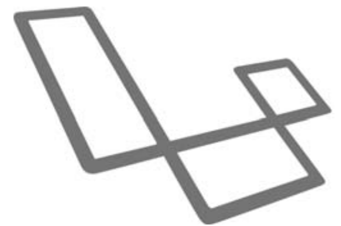
Поскольку файл pc.js хранит сценарии, используемые на разных страницах, они должны быть универсальными. Поэтому сначала мы проверяем, присутствуют ли

на странице кнопки сохранения и восстановления статьи, и лишь в случае их наличия выполняем остальной код.

Собственно, в этом коде нет ничего сложного и нового для нас. При нажатии на кнопку сохранения статьи выполняется проверка, ввел ли пользователь ключевые сведения о статье: ее заголовок и содержание. Если так, мы сохраняем в локальном хранилище заголовок, слаг, анонс, содержание статьи и обозначение подкатегории, к которой она относится.

А при нажатии кнопки восстановления статьи мы проверяем, были ли сохранены в локальном хранилище заголовок и содержание статьи (ее ключевые данные). Если так, мы выполняем чтение всех сохраненных ранее сведений о статье и подстановку их в соответствующие элементы управления.

Приведенный в листинге 45.1 код выполняет самые элементарные задачи по сохранению статей. Например, мы можем дополнить его таким образом, чтобы, например, кнопка сохранения статьи в зависимости от того, ввел ли пользователь ключевые данные, становилась доступной или недоступной.



ГЛАВА 46

Публикация Web-сайта

Что ж, наш сайт полностью готов, проверен и работоспособен. Пришло время опубликовать его в Интернете, сделав тем самым доступным для всех потенциальных читателей. Чем мы и займемся в этой, последней главе книги.

Подготовка Web-сайта к публикации

Сначала нам следует определенным образом подготовить сайт к публикации. Процесс подготовки включает в себя установку окончательных настроек и удаление ненужных файлов.

Настоятельно рекомендуется сделать копию сайта, все манипуляции по подготовке сайта выполнять над этой копией и ее же впоследствии публиковать в Сети. Код изначального сайта мы сохраним на тот случай, если потом захотим что-либо в нем переделать.

Указание окончательных настроек

Первое, что нам необходимо сделать, — задать настройки, с которыми будет работать опубликованный в Интернете сайт. Почти все эти настройки указываются в файле .env. Давайте перечислим, что нам нужно сделать.

1. Прежде всего, указать, что отныне сообщения об ошибках будут выводиться в сокращенном формате. Для этого следует задать для параметра `APP_DEBUG` значение `false`.

2. Если применяется кэширование данных, задать рабочие параметры кэширования.

   Последующие настройки имеет смысл указывать лишь непосредственно перед публикацией сайта, поскольку до этого момента они могут быть неизвестны. Обычно все эти сведения можно узнать, ознакомившись с документацией хостинг-провайдера, на мощностях которого будет опубликован сайт.

3. Записать реальный интернет-адрес публикуемого сайта, задав его в качестве значения параметра `APP_URL`.

4. Указать параметры для подключения к базе данных на компьютере хостинг-провайдера.
5. Указать параметры для отправки электронной почты, действующие на компьютере хостинг-провайдера.
6. Возможно, понадобится изменить параметры хранилищ и дисков, на которые будут сохраняться выгруженные посетителем файлы (см. *главу 32*). Эти настройки указываются в файле config\filesystem.php.

После этого рекомендуется сгенерировать единый файл настроек, который будет загружаться и обрабатываться быстрее, чем множество разрозненных файлов. Для этого нужно открыть консоль OpenServer, перейти в папку проекта и отдать команду:

```
php artisan config:cache
```

В результате в папке bootstrap\cache будут созданы файлы config.php и services.php, хранящие все настройки проекта.

И, разумеется, нужно внести правки в программный код, указав в нем реальные значения параметров. Это касается, в частности, адреса электронной почты администратора сайта (мы поместили его на главной странице) и количества позиций, выводящихся на странице пагинатора.

Удаление ненужных данных

Как мы уже знаем, шаблоны Laravel пишутся с применением встроенного языка шаблонизатора Blade (за подробностями — к *главе 31*). При первом выполнении каждого шаблона присутствующие в его коде команды Blade преобразуются в обычный PHP-код, и преобразованный таким образом файл шаблона сохраняется в папке storage\framework\views (он-то и будет использован для формирования страницы в дальнейшем).

Чтобы уменьшить объем информации, передаваемой по сети, мы можем удалить эти преобразованные шаблоны — все равно они впоследствии будут созданы повторно. Для этого достаточно открыть консоль OpenServer, перейти в папку проекта и отдать команду:

```
php artisan view:clear
```

По той же самой причине следует очистить кэш. Для этого нужно отдать в консоли команду:

```
php artisan cache:clear
```

Подсистема разграничения доступа Laravel позволяет зарегистрированным пользователям выполнить сброс пароля. При заполнении формы для отправки электронного письма, содержащего гиперссылку на страницу сброса пароля, в таблицу `password_resets` базы данных сайта заносится особый идентификатор, впоследствии используемый в процедуре сброса пароля. Эту таблицу перед публикацией сайта имеет смысл очистить, для чего достаточно отдать в консоли OpenServer команду:

```
php artisan auth:clear-resets
```

При создании нового проекта формируется большое количество папок и файлов, как говорится, на все случаи жизни. Но жизнь — штука сложная, и некоторые из сгенерированных папок и файлов в дальнейшем нам просто не понадобятся. Давайте рассмотрим папки и файлы, которые, скорее всего, окажутся не нужными для успешной работы опубликованного сайта. Их немного.

- tests — хранит файлы с кодом тестов. Мы не используем подсистему автоматического тестирования, поэтому сразу можем удалить эту папку.
- resources\assets — хранит файлы сценариев и файлы таблиц стилей, записанных на языке SASS. Мы уже создали файлы сценариев и внешние таблицы стилей, которые нам нужны, и сохранили их в папках public\css и public\js, поэтому с чистой совестью можем удалить эту папку вместе с ее содержимым.

  SASS (Syntactically Awesome Stylesheets, синтаксически удивительные таблицы стилей) — язык написания таблиц стилей, являющийся дальнейшим развитием CSS. Позволяет несколько сократить трудоемкость разработки представления для страниц. Однако, поскольку ни один Web-обозреватель не поддерживает язык SASS, написанные на нем таблицы стилей в обязательном порядке подлежат преобразованию (трансляции) в CSS с помощью особой программы-транслятора.
- database\migrations — миграции в составе уже законченного и работающего сайта совершенно не нужны. Удаляем эту папку.
- Все файлы, хранящиеся непосредственно в папке проекта, кроме .env и .htaccess, — используются только при установке дополнительных библиотек с помощью программы Composer и прототипировании различных программных модулей сайта утилитой Artisan. Как только разработка сайта закончена, их можно удалить.

  Если ранее был создан единый файл настроек проекта, также можно удалить и файл .env, поскольку нужда в нем отпадает.

  Внимание!

  В папке проекта, помимо всего прочего, также содержится исполняемый файл утилиты Artisan. После его удаления выполнить любые команды средств прототипирования не удастся.
- storage\app\public — соответствует публичному диску. Поскольку мы храним выгруженные пользователями файлы на локальном диске, можем удалить эту папку. (Хотя, в принципе, можем и не делать этого — ибо эта папка изначально пуста.)

В главах, посвященных разработке сайта, также упоминалось о файлах, которые можно безбоязненно удалить. Это файлы контроллеров и шаблонов, сгенерированные при создании нового проекта.

Создание страниц с сообщениями об ошибках

Если в настройках проекта указан вывод подробных сообщений об ошибках (то есть параметру `APP_DEBUG` файла .env присвоено значение `true`), при возникновении ошибки в коде сайта будет открыта страница с ее подробным описанием. С одной стороны, эти сведения неоценимы в процессе разработки, но, с другой, они могут сообщить злоумышленнику сведения о программной логике сайта, что совершенно неприемлемо.

Именно поэтому перед публикацией сайта обязательно нужно задать параметру `APP_DEBUG` значение `false` (что, собственно, мы уже сделали). Тогда, если возникнет программная ошибка, будет выведена страница с ее сокращенным описанием, из которого злоумышленник не сможет получить сведения о программной логике сайта, необходимые ему для совершения взлома.

Коды наиболее часто встречающихся ошибок такого рода и ситуации, в которых возникают эти ошибки, приведены в табл. 46.1.

Таблица 46.1. *Ошибки протокола HTTP*

Код ошибки	Ситуация, при которой возникает ошибка
403	Пользователь пытается добавить, изменить или удалить запись модели, не имея необходимых прав для этого
404	Выполнено обращение по несуществующему (не указанному в файле routes\web.php) маршруту
503	При выполнении программного кода возникла ошибка

Мы можем задать свои собственные шаблоны для страниц, которые будут выводиться при возникновении ошибок. Эти шаблоны должны иметь имена вида <*код ошибки*>.blade.php и храниться в папке resources\views\errors. Кстати, в этой папке уже присутствует шаблон 503.blade.php.

При разработке шаблонов страниц с сообщениями об ошибках нужно иметь в виду следующее:

- такие страницы выводятся не контроллерами нашего сайта, а программным ядром Laravel. Следовательно, внутри них недоступны значения, сохраненные в свойствах классов контроллеров, что входят в состав сайта. Применительно к нашему сайту, в коде шаблонов страниц с сообщениями об ошибках мы не можем обратиться к свойству `parentView` контроллеров, хранящему имя родительского шаблона;
- шаблонам этих страниц мы не можем передать какие-либо данные из контроллеров в составе контекста данных (что, собственно, вытекает из сказанного ранее). Однако можно передать им нужные значения в составе разделяемых данных или с применением составителей (они были описаны в *главе 31*).

Код шаблона страницы, сообщающей о возникновении ошибки 404 (как наиболее часто встречающейся), приведен в листинге 46.1.

Листинг 46.1

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="/css/basic.css" type="text/css">
  </head>
  <body>
    <h1>Ошибка</h1>
    <p>Запрашиваемая вами страница не найдена.</p>
    <p><a href="/">Главная страница</a></p>
  </body>
</html>
```

Публикация сайта

Итак, все подготовительные действия мы выполнили. Настает волнующий миг публикации сайта в Интернете и ожидания первых посетителей.

Если мы собираемся публиковать сайт на своем собственном компьютере, нам, собственно, больше ничего не нужно делать. Придется лишь соответственно изменить настройки программного пакета хостинга OpenServer (как это сделать, описано в *приложении 1*).

Если же мы собираемся опубликовать сайт на компьютере стороннего хостинг-провайдера, мы выполним следующие действия:

1. Создадим резервную копию (дамп) базы данных нашего сайта. Как это сделать, описано в *приложении 2*.
2. При регистрации нового сайта через интерфейс хостинг-провайдера автоматически создается новая база данных и новый пользователь для подключения к ней. Запомним или запишем имя базы данных, имя и пароль этого пользователя.
3. В настройках выгружаемого сайта укажем интернет-адреса самого сайта, сервера данных (их можно узнать, ознакомившись с документацией хостинг-провайдера), имя базы данных, созданной на сервере хостинг-провайдера, а также имя и пароль пользователя.
4. Также при регистрации нового сайта для него создается корневая папка. Выгрузим в нее все файлы сайта.
5. Импортируем во вновь созданную на сервере хостинг-провайдера базу данных дамп базы данных нашего сайта.
6. Наконец, проверим, работает ли опубликованный сайт.

На этом разрешите закончить книгу, посвященную HTML 5, CSS 3, PHP, MySQL и популярнейшему в мире фреймворку Laravel. И позвольте пожелать удачи в путешествии в чарующий мир Web-программирования!

Заключение

Вот мы и дочитали до конца книгу о фреймворке Laravel, а также о Web-технологиях HTML 5, CSS 3, JavaScript, MySQL и PHP, без которых в Web-программировании не обойтись. И попутно закончили разработку сайта системы электронных публикаций «СЭП».

Сайт у нас получился вполне функциональный, готовый к публикации в Интернете и, в принципе, даже к практическому использованию. Так что, если вам, уважаемые читатели, нужна интернет-система подобного рода, можете обратить внимание на этот сайт. (Благо его без проблем можно загрузить с сайта издательства.)

Так или иначе, мы получили все необходимые знания и навыки для разработки сайтов практически любого назначения. Так что, если вдруг нам потребуется сделать сайт, мы его сделаем.

А теперь — небольшое домашнее задание, направленное на совершенствование разработанного нами сайта. Автор уверен, что вы с ним справитесь.

- При выводе страницы просмотра статьи в панели навигации никак не выделяется пункт, указывающий на страницу подкатегории, к которой относится выводимая на экран статья, более того, там даже не выводится список пунктов-подкатегорий. Сделайте так, чтобы он выводился, а соответствующий пункт панели навигации выделялся — как и на странице подкатегории.
- При сохранении новой категории, подкатегории или статьи ее название (заголовок) никак не проверяется на уникальность. Сделайте так, чтобы такая проверка выполнялась, но только при создании новой категории, подкатегории или статьи (а не при правке уже существующей).
- Попробуйте реализовать поддержку тегов BBCode также и в содержаниях комментариев.

Laravel — исключительно мощный фреймворк, предоставляющий большое количество программных инструментов для выполнения самых разных задач. И далеко не все эти инструменты описаны в книге. Так, Laravel включает в свой состав мощные средства для выполнения ресурсоемких задач по расписанию — они могут пригодиться, в частности, при реализации массовой почтовой рассылки. Еще там имеются средства для отправки сообщений Web-обозревателю — если мы соберемся

делать сайт с чатом, они нам очень помогут. И, наконец, имеется возможность разработки дополнительных команд для утилиты Artisan — эти команды могут как выполнять прототипирование различных программных модулей, так и запускать какие-либо действия в уже работающем сайте (например, ту же самую почтовую рассылку).

И не следует забывать о дополнительных библиотеках, расширяющих функциональность Laravel. Одну такую библиотеку — Captcha for Laravel — мы уже рассмотрели в книге. На самом же деле их гораздо больше.

Да и технологии HTML, CSS, JavaScript, MySQL и PHP также не стоят на месте. Они продолжают развиваться, и о появляющихся в них нововведениях желательно быть в курсе.

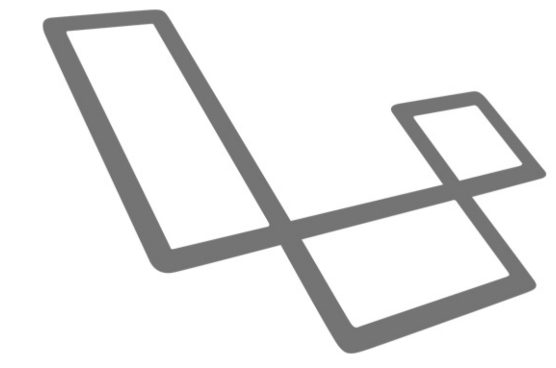
Так что нам, Web-программистам, придется учиться и в дальнейшем. Нам помогут в этом материалы, опубликованные на сайтах, список которых приведен в табл. З.1.

Таблица З.1. *Список рекомендуемых Web-сайтов*

Интернет-адрес	Описание
http://www.w3.org/	Сайт организации W3C. Все актуальные интернет-стандарты: HTML, CSS и JavaScript
http://www.w3schools.com/	Отличный сайт с руководствами и справочниками по многим современным интернет-технологиям
http://php.net/	Сайт разработчиков платформы PHP. Доступна документация на русском языке
https://www.mysql.com/	Сайт разработчиков сервера данных MySQL
https://laravel.com/	Сайт разработчиков фреймворка Laravel. Имеется довольно-таки исчерпывающая документация, ссылки на дополнительные обучающие материалы, в том числе и видеокурсы, и форум
https://laravel.ru/	Русскоязычный сайт по Laravel. Документация, переведенная на русский язык (правда, по устаревшей версии 5.2), большое количество полезных статей, форум
https://laravel-news.ru/	Другой русскоязычный сайт, посвященный Laravel. Большое количество практических уроков и полезных советов
https://vk.com/laravel_rus	Сообщество социальной сети «ВКонтакте», посвященное Laravel
https://ospanel.io/	Сайт разработчиков пакета хостинга OpenServer
https://getcomposer.org/	Сайт разработчиков утилиты Composer. Содержит исчерпывающую документацию по этому программному продукту

На этом — все. Автор прощается с вами еще раз и желает удачи в создании новых замечательных сайтов.

Владимир Дронов



ПРИЛОЖЕНИЯ

Приложение 1.	Установка и настройка пакета OpenServer
Приложение 2.	Работа с базами данных MySQL в программе phpMyAdmin
Приложение 3.	Перекодирование видеофайлов в формат MP4
Приложение 4.	Файловый архив

ПРИЛОЖЕНИЕ 1

Установка и настройка пакета OpenServer

В этом приложении мы рассмотрим установку и настройку пакета хостинга OpenServer. Этот пакет включает в себя все программы, необходимые для развертывания Web-сайта: Web-сервер Apache, программную платформу PHP, сервер данных MySQL и большое количество дополнительных приложений, включая консоль OpenServer, Composer и средство для работы с базами MySQL, называемое phpMyAdmin.

Отметим, что с недавних пор этот пакет сменил имя на Open Server Panel. Однако в книге он по-прежнему называется OpenServer.

Пакет OpenServer можно найти по интернет-адресу **https://ospanel.io/**. Автор хочет предупредить, что его дистрибутивный файл имеет размер порядка 1 Гбайт, а его загрузка производится крайне медленно — почти два часа. Вероятно, компьютер, на котором находится сайт пакета, подключен к Сети по очень медленному каналу.

Установка

После запуска дистрибутивного файла OpenServer на экране появится окно, предлагающее указать путь, по которому будет установлен пакет. По умолчанию он устанавливается в корневую папку диска C:.

> ***Внимание!***
> Не следует устанавливать OpenServer в папку, защищенную системой UAC. К таким папкам относится папка, где установлена сама Windows, а также папки *Program Files* и *Program Files (x86)*, которая присутствует лишь на 64-разрядных системах.

После нажатия кнопки **OK** начинается собственно процесс распаковки файлов пакета и копирование их по указанному нами пути. Длится он также довольно долго, поскольку пакет имеет большой объем.

Все файлы пакета при установке записываются в папку OpenServer, которая создается по указанному пути.

В папке OpenServer будут созданы следующие папки:

- domains — хранит корневые папки всех сайтов, что будут работать под управлением этого пакета.

Изначально там присутствует только папка localhost — корневая папка единственного сайта, запускаемого по умолчанию. Ее можно использовать для размещения сайта, разрабатываемого в этой книге, предварительно удалив из нее все файлы (они хранят код тестового сайта, позволяющего проверить работоспособность OpenServer сразу после его установки);

- modules — хранит все основные программы-серверы, входящие в состав пакета: Web-серверы, серверы данных, платформу PHP и проч.;
- progs — хранит вспомогательные программы, входящие в состав пакета (в частности, Web-обозреватель Google Chrome, в котором работает комплектный пакет phpMyAdmin);
- userdata — хранит рабочие данные всех программ, входящих в состав пакета.

  Здесь нам наиболее интересна папка userdata\temp\email, в которой, в виде обычных текстовых файлов, сохраняются все электронные письма, отправленные средствами Laravel. В процессе тестирования средств сброса пароля (см. *главу 33*) все письма, содержащие гиперссылки на страницу сброса пароля, будут сохранены именно там.

Запуск, перезапуск и остановка

В папке OpenServer будут находиться два исполняемых файла: Open Server x86.exe и Open Server x64.exe. Первый файл представляет собой 32-разрядную редакцию пакета, второй — его 64-разрядную редакцию.

Чтобы запустить пакет, следует выполнить запуск одного из этих файлов — в зависимости от используемой редакции Windows. Возможно, пользователи Windows Vista и более поздних версий Windows столкнутся с тем, что пакет потребует для работы повышенных прав, и будут вынуждены ответить положительно на появившееся на экране предупреждение системы UAC.

> ***Внимание!***
>
> При первом запуске OpenServer может начать установку библиотек времени исполнения Microsoft Visual C++ версий 2005–2015. Их установку обязательно следует довести до конца, иначе могут быть проблемы с входящими в состав пакета программами.

После запуска OpenServer в области уведомлений (системном трее) появится его значок, имеющий вид красного флага. Если щелкнуть на этом значке левой или правой кнопкой мыши, появится контекстное меню, с помощью которого можно запустить или остановить входящие в состав пакета программы-серверы — Web-сервер и сервер данных, а также закрыть сам пакет.

Изначально, сразу же после запуска OpenServer, все программы-серверы, что включены в его состав, неактивны. Их нужно запустить, что выполняется выбором пункта **Запустить** контекстного меню флажка.

Как только и Web-сервер, и сервер данных будут запущены, OpenServer просигнализирует об этом, изменив цвет флажка — значка хостинга — на зеленый.

Если потребуется остановить программы-серверы, следует выбрать в контекстном меню пункт **Остановить**. Пункт **Перезапустить** позволит выполнить перезапуск серверов, что может понадобиться после внесения правок в их конфигурацию. А пункт **Выход** завершает работу самого хостинга OpenServer, при этом также будут остановлены оба сервера.

Настройка

Если мы собираемся разрабатывать сайт с применением фреймворка Laravel, нам следует изменить настройки пакета. Так, нам понадобится указать использование более новой версии PHP, нежели та, что задействована в нем по умолчанию.

Выберем в контекстном меню значка-флага, что отображается в области уведомлений, пункт **Настройки**. На экране появится окно настроек хостинга, открытое на вкладке **Основные** (рис. П1.1).

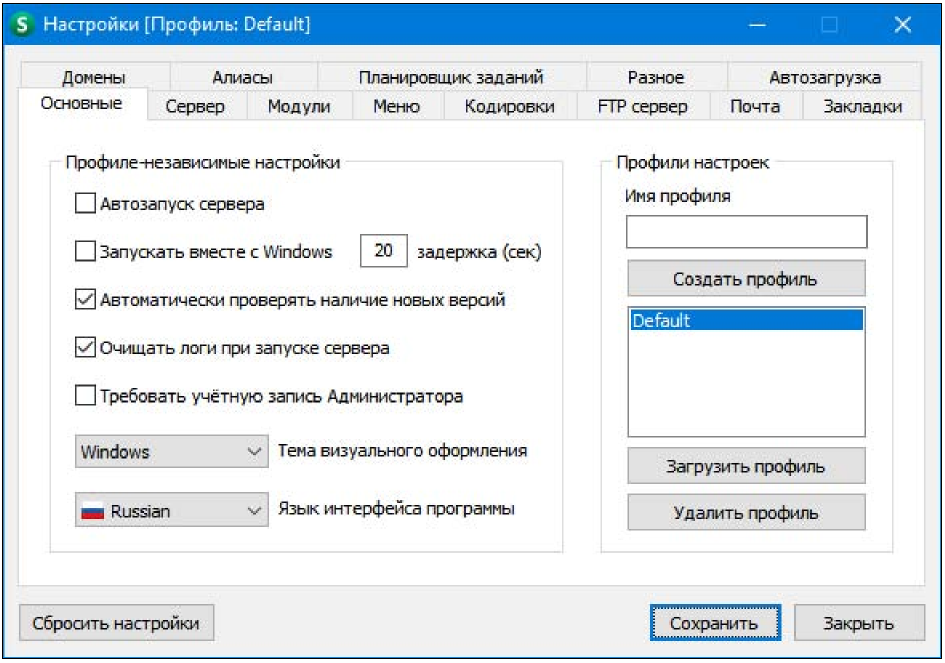


Рис. П1.1. Вкладка **Основные** окна настроек хостинга OpenServer

Здесь нас интересуют следующие элементы управления:

- флажок **Автозапуск сервера** при установке предписывает запускать оба сервера сразу при запуске самого хостинга;
- флажок **Запускать вместе с Windows** при установке указывает OpenServer запускаться сразу при загрузке системы. При этом мы можем установить задержку запуска серверов, величина которой, измеренная в секундах, заносится в расположенное правее флажка поле ввода;

- флажок **Автоматически проверять наличие новых версий** предлагает пакету сразу после запуска выполнить проверку на наличие новой версии. Если таковая обнаружится, на экране появится сообщающее об этом окно;
- раскрывающийся список **Язык интерфейса программы** позволяет выбрать язык. Впрочем, нужный язык в нем выбирается автоматически.

А теперь переключимся на вкладку **Модули** (рис. П1.2). Здесь указываются версии программ-серверов, которые следует использовать.

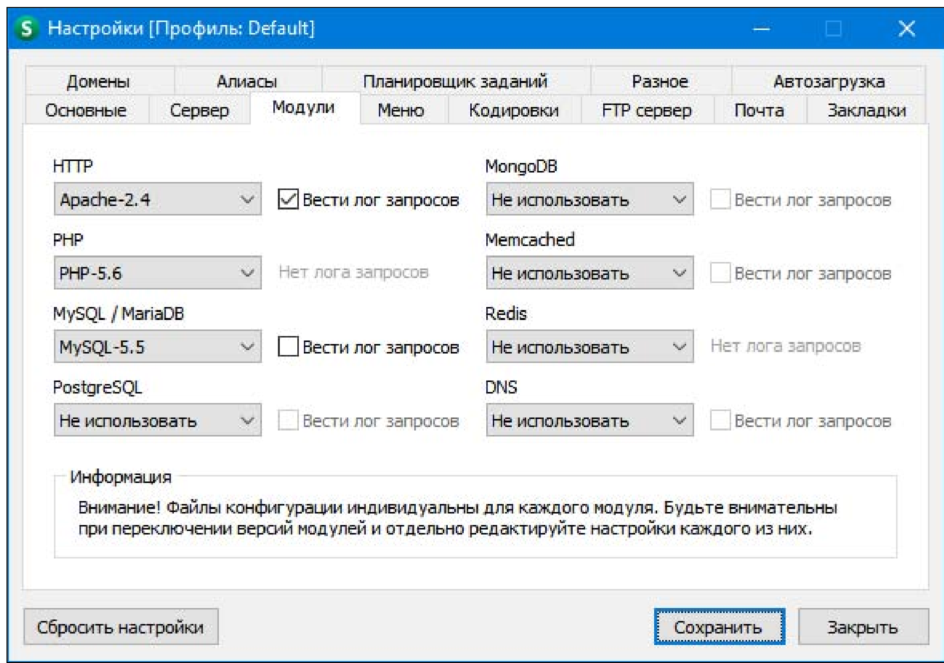


Рис. П1.2. Вкладка **Модули** окна настроек хостинга OpenServer

Поскольку Laravel требует PHP версий 5.6.4 или выше, а PHP версий 5.6 и выше может работать только под Apache версии 2.4 и выше, выбираем в раскрывающемся списке **HTTP** пункт **Apache-2.4**, а в раскрывающемся списке **PHP** — пункт **PHP-5.6**.

Задав нужные настройки, следует нажать кнопку **Сохранить**. Если серверы уже запущены, OpenServer выдаст предупреждение об их перезапуске, на которое нужно ответить положительно.

Кнопка **Закрыть** закрывает окно настроек.

Запуск консоли OpenServer

Для установки всевозможных библиотек, включая сам фреймворк Laravel, а также для запуска средств прототипирования этого фреймворка мы воспользуемся консолью OpenServer. Сейчас мы узнаем, как запустить ее.

Перед запуском консоли следует запустить сам хостинг, то есть все программы-серверы, что входят в его состав. Для этого, как мы уже знаем, нужно выбрать пункт **Запустить** контекстного меню, что появляется при щелчке правой кнопкой мыши на значке OpenServer в области уведомлений.

Сама консоль запускается выбором пункта **Консоль** подменю **Дополнительно** того же меню. Ее окно показано на рис. П1.3.

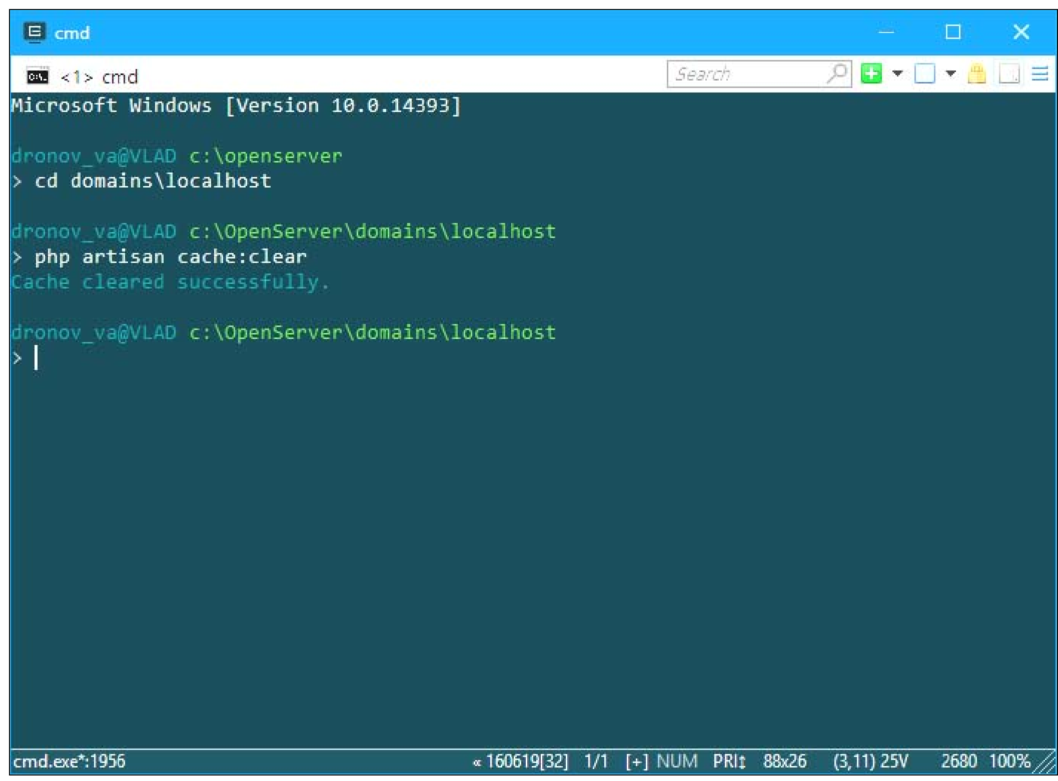


Рис. П1.3. Окно консоли OpenServer

Консоль OpenServer аналогична стандартной утилите Командная строка, что поставляется в составе Windows. Команды в ней набираются вручную и завершаются нажатием клавиши <Enter>. Поддерживается команда `cd`, выполняющая переход в другую папку, равно как и прочие команды Командной строки.

Изначально, сразу после запуска, в качестве текущей папки в консоли задана папка, в которой установлен сам пакет OpenServer (по умолчанию — C:\OpenServer).

Чтобы закрыть консоль, достаточно либо щелкнуть на кнопке закрытия ее окна, либо отдать непосредственно в консоли команду:

```
exit
```

ПРИЛОЖЕНИЕ 2

Работа с базами данных MySQL в программе phpMyAdmin

Это приложение посвящено программе phpMyAdmin — мощному инструменту для работы с базами данных формата MySQL, которую мы будем использовать для создания пользователей баз данных и самих баз, правки записей, экспорта и импорта баз данных.

phpMyAdmin поставляется в составе OpenServer. Нам не придется самим искать, загружать и устанавливать ее.

phpMyAdmin написана на PHP, представляет собой набор серверных программ и работает через обычный Web-обозреватель. Из чего следует, что для начала работы с этой программой мы должны запустить хостинг OpenServer и входящие в его состав программы-серверы.

Запуск и вход

Запустить саму phpMyAdmin можно выбором одноименного пункта подменю **Дополнительно** контекстного меню флажка, которым в области уведомлений (системном трее) представляется запущенный хостинг OpenServer. Сразу после этого запустится поставляемый в составе этого пакета Web-обозреватель Google Chrome, в окне которого и будет открыта начальная страница phpMyAdmin.

Поскольку сервер данных MySQL реализует разграничение доступа, перед началом работы с ним нам понадобится выполнить вход. Форма для входа находится непосредственно на начальной странице phpMyAdmin. Укажем для интерфейса программы русский язык (если он не был указан изначально), выбрав в раскрывающемся списке языков соответствующий пункт, введем имя пользователя и пароль и нажмем кнопку входа.

> ***Внимание!***
>
> Изначально в списке пользователей MySQL занесен пользователь с именем `root` и пустым паролем, обладающий правами администратора и могущий подключаться к серверу лишь с того компьютера, на котором установлена сама программа сервера MySQL. Для работы с любыми базами данных мы можем выполнить вход под этим пользователем.

После успешного входа мы увидим главную страницу phpMyAdmin (рис. П2.1). В ее левой части находится список всех созданных к этому моменту баз данных, а в правой будут появляться инструменты для работы с ними.

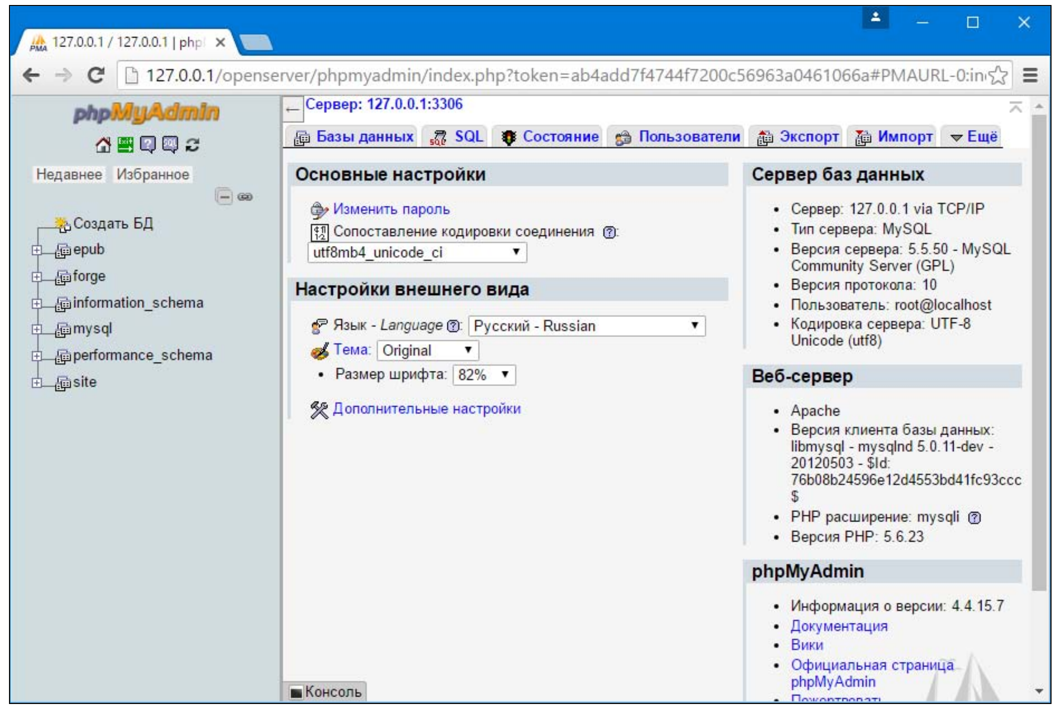


Рис. П2.1. Главная страница phpMyAdmin

Кстати, чтобы вернуться на главную страницу с любой другой страницы этой программы, достаточно щелкнуть по кнопке **К началу**, которая находится в левом верхнем углу любой страницы, над списком баз данных.

Работа с пользователями

Рассмотрим работу с пользователями MySQL: их создание (в том числе и вместе с одноименными базами данных), правку и удаление.

Создание пользователя и базы данных

Как мы уже знаем из *главы 26*, все данные сайта, написанного с использованием фреймворка Laravel, хранятся в одной базе, а подключение к этой базе всегда выполняется от имени одного и того же пользователя. Поэтому phpMyAdmin, чтобы упростить нам работу, позволяет создать сразу и нового пользователя, и базу данных, которая получит то же имя, что и сам пользователь, и на работу с которой этому пользователю будут предоставлены полные права.

Перейдем на главную страницу программы phpMyAdmin и выберем в ее правой части вкладку **Пользователи**. На экране появится страница списка пользователей (рис. П2.2).

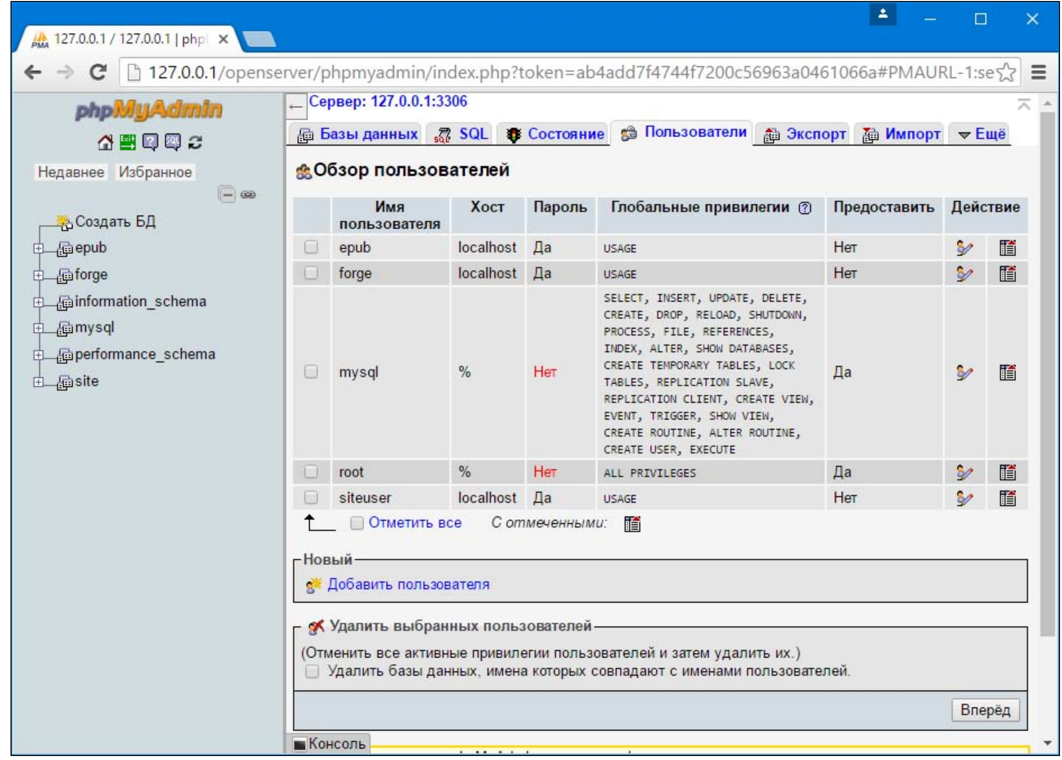


Рис. П2.2. Страница списка пользователей

На этой странице нам нужна гиперссылка **Добавить пользователя**. Щелкнем на ней и увидим страницу добавления пользователя (рис. П2.3).

В верхней части этой страницы находится форма для указания сведений о создаваемом пользователе. Укажем их.

Поскольку мы хотим, чтобы новый пользователь мог подключаться к серверу данных лишь с того компьютера, на котором установлен сам этот сервер, выберем в раскрывающемся списке **Хост** пункт **Локальный**. Пароль для пользователя необходимо задать дважды: в полях ввода **Пароль** и **Подтверждение**.

Чтобы вместе с новым пользователем создать для него и базу данных, установим флажок **Создать базу данных с таким же именем и предоставить на нее все привилегии**. Как уже говорилось, эта база данных будет называться так же, как и сам пользователь, а последний получит полные права доступа к ней.

Процесс создания нового пользователя завершается нажатием кнопки **Вперед**. Эта кнопка находится в самом низу страницы добавления пользователя (и на рис. П2.3 не показана).

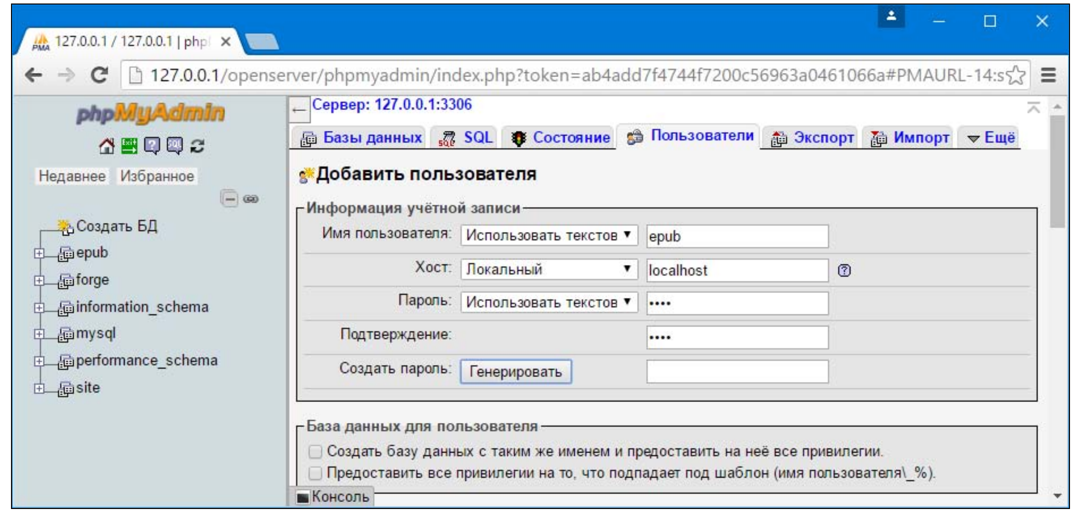


Рис. П2.3. Страница добавления пользователя

Правка и удаление пользователей

Мы можем изменить пароль у любого из уже созданных к этому моменту пользователей. Для чего перейдем на страницу списка пользователей (см. рис. П2.2), отыщем в этом списке соответствующий пункт и нажмем присутствующую в нем кнопку **Редактирование привилегий**. На странице правки пользователя, которая появится на экране, мы отыщем группу элементов управления **Изменить пароль**, введем новый пароль в поля ввода **Пароль** и **Подтверждение** и нажмем расположенную ниже кнопку **Вперед**.

Чтобы удалить ненужного или ошибочно созданного пользователя, возможно, вместе с принадлежащей ему базой данных, нужно, опять же, перейти на страницу списка пользователей. Там мы установим флажок, находящийся в крайнем левом столбце соответствующего пункта. Если мы хотим также удалить принадлежащую этому пользователю базу данных, то установим и флажок **Удалить базы данных, имена которых совпадают с именами пользователей**, находящийся в группе элементов управления **Удалить выбранных пользователей**. И, чтобы выполнить собственно удаление, нажмем находящуюся в той же группе кнопку **Вперед**. phpMyAdmin выведет предупреждение, на которое нам нужно ответить положительно.

phpMyAdmin предоставляет богатые и довольно простые в использовании средства для создания и правки структур данных: таблиц, полей, индексов и связей. Однако для этого удобнее применять миграции Laravel (см. *главу 27*).

Работа с записями таблиц

Чтобы проверить, как работают серверные программы, составляющие наш сайт, в таблицы базы данных следует занести некоторые отладочные данные. Также нам понадобится изменить роль одного из первых созданных нами пользователей, чтобы получить хотя бы одного администратора. Сейчас мы выясним, как это делается.

Обратим внимание на левую часть страницы. Там выводится иерархический список уже созданных к данному моменту баз данных — каждая из них представляется отдельной «ветвью» этого списка. Щелкнув на значке «плюс», что находится левее названия «ветви», мы развернем ее и увидим список пунктов, представляющих таблицы, что входят в состав этой базы (рис. П2.4).

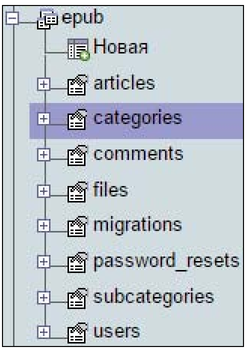


Рис. П2.4. «Ветвь» расположенного слева иерархического списка, представляющая базу данных, и ее пункты, что представляют таблицы, входящие в состав этой базы

Щелкнем на пункте, который представляет нужную нам таблицу базы данных. В правой части страницы появятся сведения об этой таблице (рис. П2.5) — как видим, на ней присутствует также и список уже имеющихся в ней записей.

Добавить новые записи в таблицу можно на странице добавления записей (рис. П2.6). Перейти на нее можно, выбрав вкладку **Вставить**.

Здесь мы видим две группы элементов управления, предназначенные для занесения содержимого двух добавляемых записей. (Мы можем одновременно добавить в таблицу лишь две записи, но это не критическое ограничение.) В левой части каждой группы указаны имена и параметры добавляемых полей, а в правой части находятся поля ввода для указания их значений. Отметим, что такие поля ввода существуют для всех полей таблицы, даже для поля номера, значение в которое заносится автоматически, — понятно, что значения для полей такого рода заносить вручную не нужно.

Если мы хотим добавить в таблицу лишь одну запись, то не будем заносить данные для второй записи и проверим, установлен ли флажок **Игнорировать**, находящийся над второй группой.

Занеся все нужные данные, нажмем кнопку **Вперед**, что находится в правом нижнем углу соответствующей группы. Если мы собираемся сразу же после этого

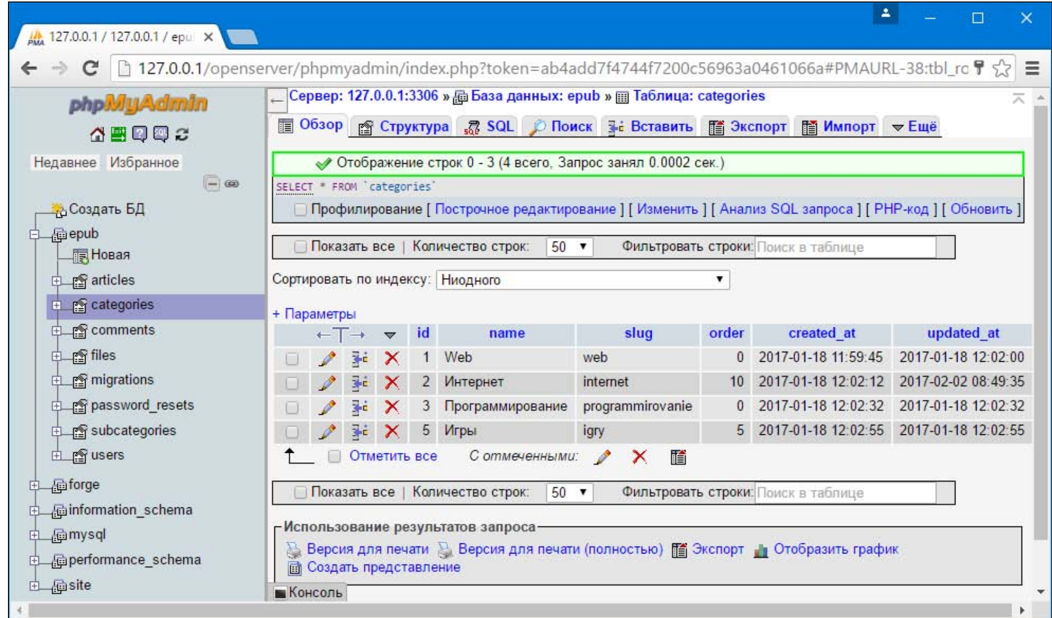


Рис. П2.5. Страница сведений о выбранной таблице

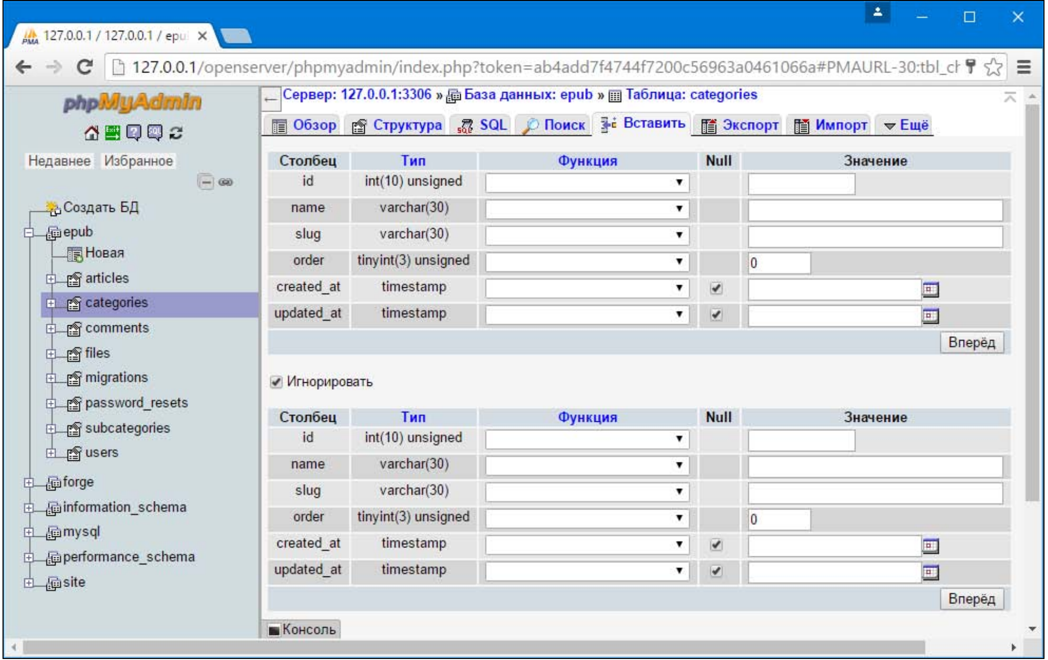


Рис. П2.6. Страница добавления записей

добавить в таблицу другие записи, то перед нажатием кнопки выберем в расположенном в правом нижнем углу страницы раскрывающемся списке **и затем** пункт **Добавить новую запись**.

Просмотреть содержимое таблицы можно на странице сведений о таблице (см. рис. П2.5). Перейти на нее можно, выбрав вкладку **Обзор**.

Любую запись таблицы мы можем исправить, щелкнув на расположенной в представляющей ее строке списка записей кнопке **Изменить** . После этого на экране появится страница правки записи, похожая на страницу добавления записей (см. рис. П2.6). Внесем необходимые правки и нажмем кнопку **Вперед**.

Удалить любую запись можно, воспользовавшись кнопкой **Удалить** , которая также находится в соответствующей нужной записи строке списка. После этого на экране появится окно-предупреждение, в котором нам следует нажать кнопку **OK**.

Перенос содержимого из одной базы данных в другую

Когда мы будем публиковать свой сайт на компьютере стороннего хостинг-провайдера, то столкнемся с необходимостью переноса данных, хранящихся в разработанной нами базе, в базу данных, созданную для нас провайдером. Или, говоря другими словами, выполнения операций экспорта и импорта данных.

> **Внимание!**
> Выполнять перенос данных следует только в пустую, не содержащую ни одной таблицы базу данных.

Экспорт данных

В процессе *экспорта* данные, хранящиеся в базе, преобразуются в другой формат и сохраняются в отдельном файле, который носит название *дампа*. Из этого файла мы впоследствии восстановим данные, сохранив их в другой базе, которая, возможно, работает под управлением другого сервера.

Экспортировать данные из базы в файл дампа можно на странице экспорта phpMyAdmin. Перейти на нее можно, щелкнув в левом списке на самой «ветви», представляющей нужную базу, и переключившись на вкладку **Экспорт**.

Операция экспорта запускается нажатием кнопки **Вперед**, находящейся в самом низу этой страницы. По ее завершению (а продолжаться она может достаточно долго, если исходная база хранит большой объем данных) Web-обозреватель предложит нам загрузить файл дампа.

Файл дампа имеет имя, совпадающее с именем исходной базой данных, и расширение sql. Он представляет собой обычный текстовый файл, который можно открыть в любом простейшем текстовом редакторе.

Импорт данных

Импорт заключается в восстановлении данных, хранящихся в файле дампа, в целевую базу. При этом в целевой базе будут созданы все необходимые структуры: таблицы, индексы и связи.

Импорт выполняется на особой странице импорта. Чтобы перейти на эту страницу, мы щелкнем на «ветви» левого списка, представляющей целевую базу, и выберем вкладку **Импорт**.

В группе элементов управления **Импортируемый файл** этой страницы мы установим переключатель **Обзор вашего компьютера**, нажмем кнопку **Выберите файл** и выберем нужный файл дампа в появившемся на экране стандартном диалоговом окне открытия файла. После чего можно нажать кнопку **Вперед**, чтобы запустить операцию импорта.

Когда импорт будет завершен, phpMyAdmin выведет страницу с соответствующим сообщением.

Выход

Закончив работу с базами данных MySQL, следует выполнить операцию выхода. Сделать это очень просто — достаточно найти над иерархическим списком баз данных, что располагается слева, кнопку **Выход** и щелкнуть на ней.

После выхода phpMyAdmin вновь выведет начальную страницу входа. Мы можем либо закрыть окно Google Chrome, в котором выполняется работа с phpMyAdmin, либо снова выполнить вход.

ПРИЛОЖЕНИЕ 3

Перекодирование видеофайлов в формат MP4

В настоящее время для публикации аудиофайлов в Интернете используется популярнейший формат MP3. Найти файлы в этом формате не составляет никакого труда — более того, если мы загрузим из Интернета какой-либо аудиофайл, то, скорее всего, он окажется в формате MP3.

Другое дело — видеофайлы. Публикуемое в Сети видео должно быть сохранено в формате MP4. Следовательно, нам придется либо поискать видеофайл, уже сохраненный в этом формате (что является серьезной проблемой), либо перекодировать его самостоятельно.

В этом нам поможет утилита с «говорящим» названием Free HTML5 Video Player And Converter. Она совершенно бесплатна и исключительно проста в использовании, а загрузить ее можно со страницы **http://www.dvdvideosoft.com/products/dvd/Free-HTML5-Video-Player-And-Converter.htm**.

> ***Внимание!***
>
> По указанному интернет-адресу находится Web-установщик, который после запуска загружает дистрибутивный комплект утилиты из Сети. Поэтому перед началом установки проверьте, соединен ли ваш компьютер с Интернетом.
>
> В процессе установки утилиты установщик спросит, желаете ли вы установить заодно и Web-обозреватель Opera. Если таковой вам не нужен, откажитесь.

Запустить установленную утилиту мы можем, найдя в меню **Пуск** группу **DVDVideoSoft** и выбрав имеющийся в ней пункт **Free HTML5 Video Player And Converter**.

Главное окно утилиты Free HTML5 Video Player And Converter показано на рис. П3.1. Рассмотрим его.

Верхнюю половину окна занимает список перекодируемых видеофайлов. (Эта утилита позволяет перекодировать за один раз произвольное количество файлов.) Для каждого файла выводится путь и продолжительность воспроизведения, если в списке присутствует несколько файлов, также выводится общая продолжительность их воспроизведения.

Чтобы добавить в список файлы, нужно нажать кнопку **Добавить файлы** и выбрать их в появившемся на экране стандартном диалоговом окне открытия файла

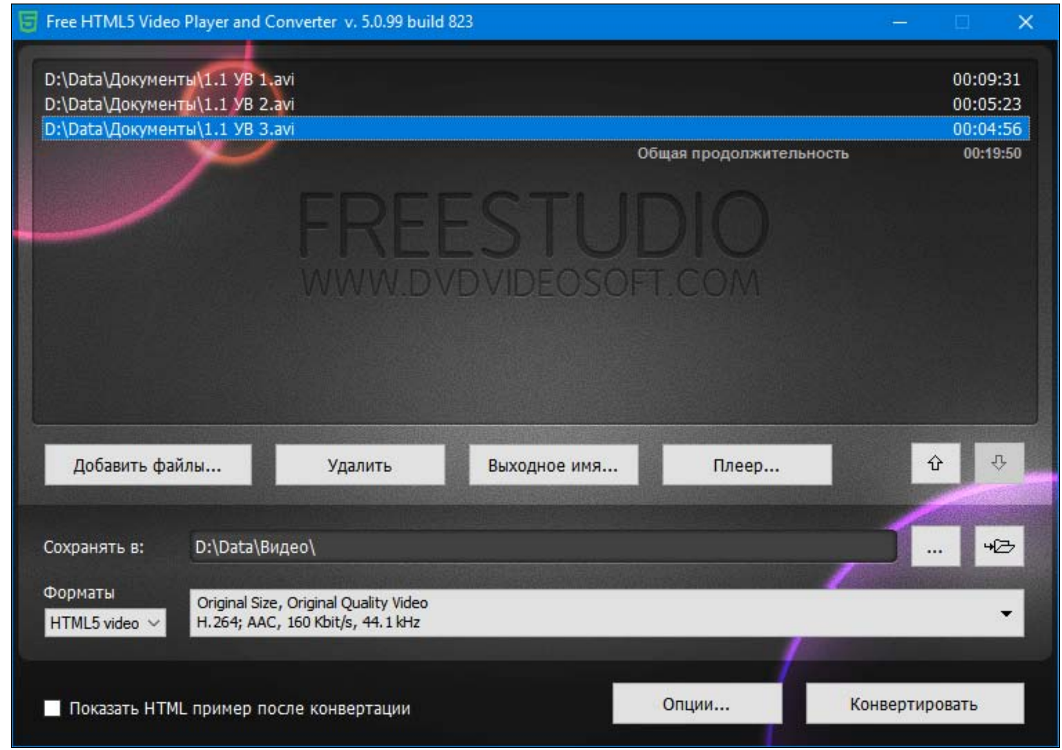


Рис. П3.1. Главное окно утилиты Free HTML5 Video Player And Converter

Windows. Также можно просто перетащить файлы из окна Проводника в окно утилиты.

Удалить из списка ошибочно добавленный в него файл можно, выбрав его и нажав кнопку **Удалить**. Отметим, что файл будет удален сразу же, без всякого предупреждения.

Поле ввода **Сохранять в** служит для указания папки, в которой будут сохранены результирующие файлы. По умолчанию это системная папка Мои видеозаписи или Видео, но мы можем выбрать любую другую. Для этого достаточно нажать расположенную правее поля ввода кнопку со значком многоточия и выбрать нужную папку в появившемся на экране стандартном диалоговом окне выбора папки.

Ниже находятся два раскрывающихся списка **Форматы**. Левый, маленький, список для нас бесполезен, так как содержит всего один пункт **HTML5 video**. А вот правый, показанный на рис. П3.2, позволяет нам выбрать параметры результирующего видеофайла (видеофайлов): его разрешение, битрейт видео, битрейт и частоту дискретизации звука.

- Пункт **Original Size, Original Quality Video** указывает перекодировать файл с изначальными параметрами, то есть без изменения разрешения и битрейтов звука и видео.

  Вероятно, в большинстве случаев следует выбирать именно этот пункт. Тем более что он выбран изначально.

- Прочие пункты задают изменение разрешения и битрейтов в процессе перекодирования.

  Нужно иметь в виду, что размер результирующего видеофайла напрямую зависит от его разрешения и битрейта: чем больше значения этих параметров — тем больше получится файл.

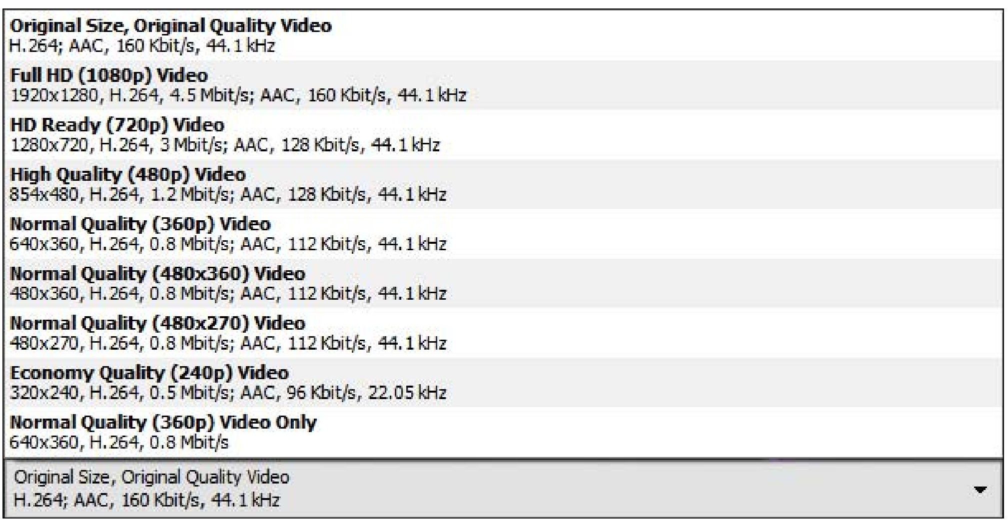


Рис. П3.2. Раскрывающийся список для выбора параметров результирующего видео

Также следует иметь в виду, что утилита Free HTML5 Video Player And Converter, помимо собственно перекодирования файлов, также создает для каждого из них Web-страницу со своего рода мультимедийным проигрывателем, в котором и будет воспроизводиться этот файл. И это поведение мы, к сожалению, отменить не можем. Однако мы можем сбросить установленный по умолчанию флажок **Показать HTML-пример после конвертации**, чтобы утилита не вывела эту страницу на экран по окончании перекодирования файлов.

Процесс перекодирования запускается нажатием кнопки **Конвертировать**. На экране появится небольшое окно, показывающее его протекание (рис. П3.3). Нажав кнопку **Остановить**, мы можем прервать его.

> ***Внимание!***
>
> Перекодирование будет прервано сразу после нажатия кнопки **Остановить** без всякого предупреждения.

Как только перекодирование всех файлов закончится, в окне процесса кнопка **Остановить** сменится кнопкой **Закрыть**. Нажмем ее, чтобы закрыть это окно.

В окне процесса, равно как и в главном окне утилиты, присутствует кнопка . Нажав ее, мы откроем окно Проводника, в котором будет показано содержимое папки с результирующими файлами — той самой, чей путь задан в поле ввода **Сохранять в**.

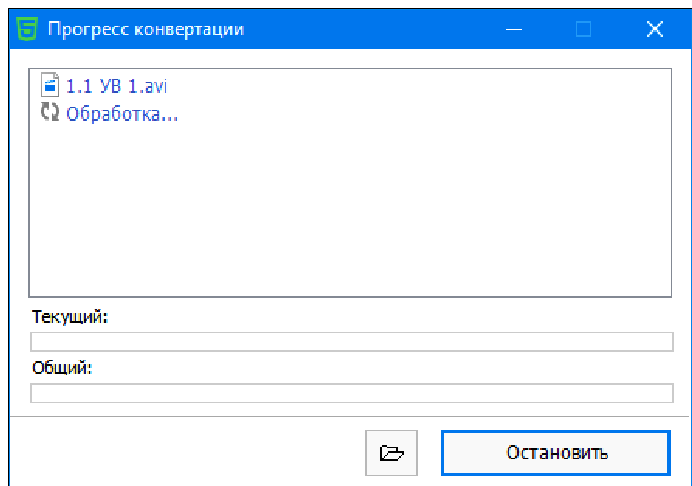


Рис. П3.3. Окно, демонстрирующее процесс перекодирования

Для каждого из перекодируемых файлов в этой папке будет создана папка с именем вида <*имя перекодируемого файла без расширения*>.html5-video. В ней находится папка video, в которой и будет сохранен результирующий файл.

Нужно отметить, что, помимо файла формата MP4, также будет создан файл формата OGG с расширением ogv (OGG video). Это поведение утилиты мы также отменить не можем. Однако мы можем просто удалить ненужные файлы.

ПРИЛОЖЕНИЕ 4

Файловый архив

По ссылке **ftp://ftp.bhv.ru/9785977538459.zip** можно загрузить электронный архив, содержащий исходные коды разработанного и рассмотренного в книге готового полнофункционального Web-сайта электронных публикаций. Эта ссылка доступна также со страницы книги на сайте **www.bhv.ru**.

Выполните распаковку содержимого электронного архива в любую папку, после чего перенесите содержимое получившейся после распаковки папки epub в папку *<папка, в которую установлен OpenServer>*\domains\localhost. (Пакет хостинга Open-Server уже должен быть установлен.)

Список файлов и папок, входящих в состав архива:

- epub — хранит все файлы и папки сайта;
- data — хранит файл дампа базы данных epub.sql;
- readme.txt — файл, обязательный для прочтения. В нем представлены необходимые сведения о развертывании сайта.

Предметный указатель

!

!doctype 23

$this 337

@font-face 114
@keyframes 189
@media 199

A

a 56
abbr 34
abort 298, 299
accept 75, 274
accessKey 60, 271, 274
action 67, 272, 449, 460, 524
add 277, 533
addColorStop 696
addEventListener 259
address 30
addSelect 434
after 377
AJAX 308, 443
AJAX-навигация 318
alert 293
align-content 159
align-items 158
align-self 160
all 423, 425, 462
allFiles 501
allows 509
alt 42, 279
altKey 263, 264
anchors 246
animation 192
animation-delay 190
animation-direction 191
animation-duration 190
animation-fill-mode 192
animation-iteration-count 190
animation-name 190
animation-play-state 191
animation-timing-function 190
any 406, 463
appendChild 255
appends 441
appName 292
AppServiceProvider 399
appVersion 292
arc 691
area 62
array 231, 334
article 31
aside 32
assign 290
associate 488
async 238
attach 488
attributes 397
audio 43
auth 515
authorize 513
AuthServiceProvider 509
autocomplete 67, 272, 274
autofocus 68, 274
autoplay 43, 281
availHeight 291
availWidth 291
average 436
avg 358, 436

B

back 291, 448
backface-visibility 180
background 128
background-attachment 127
background-clip 123
background-color 123
background-image 124
background-origin 127
background-position 126
background-repeat 125
background-size 125
base_path 444
BaseController 419
BBCode 645
beforeunload 265, 288
beginPath 690
belongsTo 391
belongsToMany 393
bezierCurveTo 692
bigIncrements 375
bigInteger 375
binary 376
bind 415
blockquote 29
Blueprint 374
blur 271, 277, 288
body 22, 246
BOM 285
boolean 230, 376
border 138
border-bottom 137

border-bottom-color 137
border-bottom-left-radius 138
border-bottom-right-radius 138
border-bottom-style 135
border-bottom-width 136
border-collapse 145
border-color 137
border-left 137
border-left-color 137
border-left-style 135
border-left-width 136
border-radius 138
border-right 137
border-right-color 137
border-right-style 135
border-right-width 136
border-spacing 144
border-style 136
border-top 137
border-top-color 137
border-top-left-radius 138
border-top-right-radius 138
border-top-style 135
border-top-width 136
border-width 137
bottom 153, 250
box-shadow 142
box-sizing 140
br 35
bubbles 267
Builder 427
button 263

C

cache 532–534
calc 102
can 513
cancellable 268
cannot 513
canplay 282
canplaythrough 282
cant 513
canvas 687
CanvasGradient 696
CanvasPattern 699
CAPTCHA 526
Captcha for Laravel 526
captcha_img 529
captcha_src 529
CaptchaServiceProvider 527
caption 49
caption-side 148
change 278, 382
char 375
charCode 264
check 515
checked 71, 273
checkValidity 277
childNodes 244
children 244
cite 34
class 94, 401
className 248
clear 150
clearInterval 295
clearRect 688
clearTimeout 295
click 262
clientHeight 248
clientWidth 248
clientX 262
clientY 262
clip 154, 706
cloneNode 256
close 288
closed 286
closePath 690
code 34
col 52
colgroup 51
color 105
colorDepth 292
colspan 54
column-count 164
column-gap 165
column-rule 166
column-rule-color 165
column-rule-style 165
column-rule-width 165
column-span 166
column-width 165
complete 279
composer 364, 472
composers 472
confirm 293
content 21, 119
contextmenu 262
Controller 419
ControllerMiddleware 450
controls 43, 281
cookieEnabled 292
coords 62
count 334, 357, 436, 441, 458, 463
counter-increment 120
counter-reset 121
create 375, 477, 487
created 398
CREATED_AT 390
createElement 254
createLinearGradient 696
createMany 488
createPattern 699
createRadialGradient 698
createTextNode 255
creating 398
csrf_field 461
CSS 83
CSSStyleDeclaration 251
ctrlKey 263, 264
current 460
currentPage 441
currentSrc 281
currentTarget 261
currentTime 280
cursor 118
customError 275

D

Data URL 298
date 230, 376
date_format 638
dateTime 376
dateTimeTz 376
DB 437
dblclick 262
dd 29
decimal 376
decrement 499, 533
default 377
defaultChecked 274
defaultPrevented 269
defaultSelected 276
defaultValue 273
defer 237
define 331, 509
del 34
delete 405, 481, 499, 503
deleted 399
deleting 399
deltaX 263
deltaY 263
denies 510

depth 458
destroy 481
detach 489
detail 263
dfn 34
DIRECTORY_SEPARATOR 444
disabled 68, 273
disassociate 488
disk 504
display 117
distinct 434
div 31
dl 29
document 241, 286
documentElement 246
doesntHave 431
DOM 240
domain 247
DOMError 298
double 376
down 374
download 447
drawImage 700
drop 384
dropForeign 383
dropIfExists 384
dropIndex 383
dropPrimary 383
dropTimestamps 384
dropTimestampsTz 384
dropUnique 383
dt 29
duration 280
durationchange 282

E

elements 272
em 34
empty-cells 148
encodeURIComponent 310
enctype 67, 272
ended 281, 283
enum 375
error 280, 283, 289, 298, 299
Event 261
eventPhase 268
except 423, 451
exec 305
exists 423, 439, 446, 503

F

fails 491
fieldset 78
figcaption 32
figure 32
file 297, 448, 501
FileList 296
FileReader 298
files 274
fill 480, 690
fillable 389
fillRect 688
fillStyle 689
fillText 695
find 424
findMany 424
findOrFail 424
findOrNew 485
first 377, 458, 462
firstChild 245
firstElementChild 245
firstItem 441
firstOrCreate 486
firstOrFail 425
firstOrNew 486
flex 163
flex-basis 162
flex-direction 156
flex-grow 162
flex-shrink 162
flex-wrap 156
float 149, 376
floor 679
Fluent 375
flush 536
focus 271, 277, 288
font 108, 695
font-family 104
font-size 105
font-style 106
font-variant 106
font-weight 106
footer 32
for 78
foreign 379
forever 532
forget 535
form 66, 273
FormRequest 496
forms 246
forward 291
fragment 441
from 524
full 460
fullUrl 443
fullUrlWithQuery 443
Function 231

G

Gate 509
get 405, 425, 462, 533
getAttribute 251
getBoundingClientRect 250
getClientOriginalExtension 502
getClientOriginalName 502
getComputedStyle 252
getContext 687
getController 473
getDate 230
getElementById 241
getElementsByClassName 243
getElementsByTagName 242
getFullYear 230
getItem 708
getMimeType 502
getMonth 230
getRouteKeyName 398
getSize 502
GET-параметр 66
GIF 40
◊ анимированный 40
global 306
globalAlpha 689
globalCompositeOperation 705
go 291
greeting 524
group 416
groupBy 437
guarded 389
guest 516

H

has 423, 431, 463, 534
hasAttribute 251
hasColumn 385
hasColumns 385
hasFile 501
hash 271, 290

hashchange 265, 288
hasMany 391
hasManyThrough 395
hasMorePages 441
hasOne 393
hasPages 441
hasTable 385
having 437
havingRaw 438
head 22, 246
header 32, 443
height 139, 250, 292, 687
history 290
History 290
home 449
host 271, 289
hostname 271, 290
hr 33
href 56, 62, 86, 270, 289
HTML 18, 23
HTMLDocument 239
HTMLElement 240
htmlentities 647
HTTP 16
http-equiv 20
HTTPS 443

I

id 64, 78, 248, 516
iframe 79
ignoreCase 306
Image 699
images 246
img 41
implode 439
in_array 484
increment 498, 533
increments 375
index 276, 306, 378, 458
indexOf 584
innerHeight 286
innerHTML 253
innerWidth 286
input 68, 278, 306, 422
inRandomOrder 433
ins 34
insert 497
insertBefore 255
insertGetId 498
instanceof 232
integer 375
ip 443
ipAddress 376
IP-адрес 16
is 443
is_array 639
is_string 639
isEmpty 441, 463
isEqualNode 257
isMethod 443
isNaN 279
isset 628
iteration 458

J

JavaScript 203
join 434
JPEG 40
JSON 314, 376, 446
justify-content 157

K

kbd 34
KeyboardEvent 264
keyCode 264
keydown 264
keypress 264
keyup 264

L

label 73, 75, 78, 276
last 458
lastChild 245
lastElementChild 245
lastIndex 306
lastItem 441
lastModified 247
lastModifiedDate 297
lastPage 442
latest 433
left 153, 250
legend 78
length 230, 231, 272, 274, 708
LengthAwarePaginator 442
lengthComputable 283
letter-spacing 107
li 28
limit 438
line 524
linear-gradient 129
lineCap 694
line-height 107
lineJoin 694
lineTo 691
lineWidth 689
link 86
links 246, 464
list-style 111
list-style-image 111
list-style-position 111
list-style-type 109
load 265, 280, 282, 288, 299
loaded 283
loadeddata 282
loadedmetadata 282
loadend 299
loadstart 282, 299
localStorage 708
location 289
Location 289
logout 517
longText 375
loop 43

M

macAddress 376
MailMessage 524
make 490
many 534
map 62
margin 135
margin-bottom 134
margin-left 134
margin-right 134
margin-top 134
mark 33
match 307, 406
Math 229
max 70, 273, 358, 436
max-height 139
maxlength 69
maxLength 273
max-width 139
media 195
mediumIncrements 375
mediumInteger 375
mediumText 375
Memo 356
message 289
MessageBag 462
meta 23

metaKey 263
method 67, 272, 443
method_field 461
middleware 409, 450
Migration 373
min 70, 273, 358, 436
min-height 139
min-width 139
miterLimit 694
model 388, 414
mousedown 262
mouseenter 262
MouseEvent 262
mouseleave 262
mousemove 262
mouseout 262
mouseover 262
mouseup 262
moveBy 288
moveTo 288, 690
multiline 306
multiple 73, 274
muted 44, 280
MySQL 348

N

name 62, 67, 68, 272, 273, 286, 297, 298, 409
naturalHeight 279
naturalWidth 279
nav 32
navigator 292
networkState 281
nextElementSibling 246
nextPageUrl 441
nextSibling 246
Node 244
nodeName 248
nodeType 248
nodeValue 248
nohref 62
Notification 522
notify 525
noValidate 272
null 226
nullable 377
nullableTimestamps 376
Number 230

O

Object 231
observe 401
offset 439
offsetHeight 249
offsetLeft 249
offsetParent 249
offsetTop 249
offsetWidth 249
ol 28
old 462
oldest 433
on 379
onDelete 380
onLine 292
only 423, 451
onUpdate 380
opacity 105
open 287, 309
opener 286
optgroup 75
option 73
options 274
order 164
orderBy 432
orderByRaw 438
orHas 432
orHaving 438
orHavingRaw 438
orWhere 429
orWhereBetween 431
orWhereColumn 431
orWhereDate 431
orWhereHas 432
orWhereIn 431
orWhereNotBetween 431
orWhereNotIn 431
orWhereNotNull 431
orWhereNull 431
outerHeight 286
outerWidth 286
outline 143
outline-color 143
outline-style 143
outline-width 143
overflow 140
overflow-x 141
overflow-y 141
ownerDocument 246

P

p 25
padding 134
padding-bottom 133
padding-left 133
padding-right 133
padding-top 133
page-break-after 200
page-break-before 200
page-break-inside 200
pageX 262
pageXOffset 285
pageY 262
pageYOffset 285
paginate 442
Paginator 440
parent 458
parentElement 244
parentNode 244
parse 314
parseInt 253
passes 491
patch 405
path 443
pathname 271, 290
pattern 69, 273, 408
patternMismatch 275
patterns 408, 411
pause 282, 283
paused 281
perPage 441
perspective 177
perspective-origin 179
PHP 330, 347
PI 228, 691
placeholder 69, 273
platform 292
play 282
playbackRate 281
playing 283
pluck 439
PNG 40
port 271, 290
position 152
post 405
poster 44, 281
pre 30
preg_match 587
preg_replace 648
preload 44, 281

preventDefault 268
previous 460
previousElementSibling 246
previousPageUrl 441
previousSibling 246
primary 378
primaryKey 390
print 287
progress 282, 299
ProgressEvent 283
prompt 293
protocol 270, 289
public_path 444
pull 535
push 277
put 405, 532
putMany 532

Q

q 34
quadraticCurveTo 692
query 423
querySelector 243
querySelectorAll 244

R

radial-gradient 130
rangeOverflow 275
rangeUnderflow 275
ratechange 283
raw 437
readAsDataURL 298
readAsText 298
readonly 68
readOnly 273
readyState 247, 281, 298, 313
readystatechange 312
rect 693
redirect 448, 449
Redirector 448
RedirectResponse 448
references 379
referrer 247
RegExp 304
rel 86
relatedTarget 262
reload 290
remaining 458
remember 535
rememberToken 377
remove 277
removeAttribute 251
removeChild 256
removeEventListener 260
removeItem 708
rename 382
repeating-linear-gradient 132
repeating-radial-gradient 132
replace 290, 307
replaceChild 256
request 422, 423, 442
required 68, 273
reset 272
resize 265, 288
resizeBy 288
resizeTo 288
resource 410
response 446
responseText 312
restore 702
restored 399
restoring 399
result 298
RGB
◊ десятичный 103
◊ десятичный с каналом прозрачности 103
◊ шестнадцатеричный 103
right 153, 250
root 443
rotate 176, 703
rotateX 178
rotateY 178
rotateZ 178
route 448, 459, 473
Route 405, 407
RouteServiceProvider 408
rowspan 54

S

s 34
samp 34
SASS 713
save 478, 487, 702
saved 399
saveMany 487
saving 399
scale 176, 704
scale3d 178
scaleX 175
scaleY 175
scaleZ 178
Schema 374
screen 291
Screen 291
screenX 262, 286
screenY 262, 286
script 236
scroll 265, 288
scrollBy 286
scrollHeight 249
scrollLeft 249
scrollTo 286
scrollTop 249
scrollWidth 249
scrollX 285
scrollY 285
sear 535
search 271, 290, 306
section 32
secure 443
seeked 283
seeking 281, 283
select 73, 277, 278, 433
selected 73, 276
selectedIndex 274
selectionEnd 273
selectionStart 273
self 338
send 310
session 463
sessionStorage 708
setAttribute 251
setCustomValidity 278
setInterval 294
setItem 708
setPath 441
setRequestHeader 310
setTimeout 295
shadowBlur 701
shadowColor 701
shadowOffsetX 701
shadowOffsetY 701
shape 62
share 471
shiftKey 263, 264
simplePaginate 440
size 73, 274, 297
skew 176
skewX 176
skewY 176

skip 439
smallIncrements 375
smallInteger 375
sometimes 492
source 306
span 35, 51
SQL 353
sqrt 229
src 41, 43, 237, 279, 280, 699
stalled 283
status 313
step 70, 273
stepMismatch 275
stopPropagation 268
Storage 503, 708
storage_path 444
store 502, 536
storeAs 503
str_replace 649
string 230, 375
stroke 690
strokeRect 688
strokeStyle 688
strokeText 695
strong 34
strtolower 587
style 87, 88, 251
subject 524
submit 272
substr 230, 655
substring 682
sum 358, 436
suspend 283
SVG 40
sync 485

T

tabindex 60, 271, 274
table 46, 381, 390
table-layout 147
tagName 248
take 438
target 57, 261, 270, 272
tbody 50
TCP-порт 17
td 47
test 305
text 276, 375
text-align 108
textAlign 695
textarea 73
textBaseline 696
textContent 253
text-decoration 106
text-indent 108
TextNode 240
text-overflow 141
TextRectangle 250
text-shadow 112
text-transform 107
tfoot 50
th 47
thead 50
time 376
timestamp 261, 376
timestamps 376, 390
timestampsTz 377
timestampTz 376
timeTz 376
timeupdate 283
tinyInteger 375
title 23, 247, 248
toArray 484
toLocaleDateString 297
toLowerCase 248
tooLong 275
top 153, 250
toString 231
total 283, 442
tr 46
transform 174
transform-origin 182
transform-style 182
transition 186
transition-delay 185
transition-duration 184
transition-property 184
transition-timing-function 185
translate 175, 702
translate3d 178
translateX 175
translateY 175
translateZ 178
truncate 499
type 68, 86, 261, 274, 297
typeMismatch 275

U

ul 28
unique 378
unload 265, 289
unsigned 377
unsignedBigInteger 376
unsignedInteger 376
unsignedMediumInteger 376
unsignedSmallInteger 376
unsignedTinyInteger 375
up 374
update 498
updated 399
UPDATED_AT 390
updateOrCreate 486
updating 398
UploadedFile 501
url 441, 443, 460
URL 247
UrlGenerator 460
usemap 62
user 515
userAgent 292
UTF-8 23
uuid 376

V

valid 275
validate 489, 491
ValidatesRequests 489
validationMessage 274
Validator 490
validity 274
ValidityState 274
value 69, 73, 273, 276
valueMissing 275
var 34
vertical-align 108, 144
video 44
videoHeight 280
videoWidth 280
view 445
View 445, 446, 471
visibility 116
volume 280
volumechange 283

W

W3C 61
waiting 283
Web-сайт
◊ динамический 324
◊ клиентская часть 13
◊ статический 14
Web-сервер 15

Web-страница
◊ главная 17
◊ динамическая 324
◊ название 23
◊ серверная 326
◊ статическая 14
◊ текущая 56
◊ целевая 56
Web-сценарий 203
◊ внешний 237
◊ внутренний 236
Web-форма 65
wheel 262
WheelEvent 263
where 407, 427
whereBetween 429
whereColumn 430
whereDate 430
whereDay 430
whereDoesntHave 432
whereHas 432
whereIn 429
whereMonth 430
whereNotBetween 429
whereNotIn 429
whereNotNull 430
whereNull 429
whereYear 430
which 263, 264
white-space 112
width 139, 250, 292, 687
willValidate 275
window 285
with 445, 524
withCount 435
withErrors 491
withInput 491
WOFF 114
word-spacing 107
word-wrap 113
write 254
writeln 254

X

XML 309
XMLHttpRequest 309

Z

Z-индекс 154

А

Абзац 25
Анимация
◊ обратная 187
◊ покадровая 188
◊ трансформационная 183
Аргумент 223
Атрибут стиля 84
◊ важный 103
◊ значение 84
◊ имя 84
Атрибут тега 20
◊ novalidate 67
◊ без значения 43
◊ значение 20
◊ имя 20
◊ необязательный 21
◊ обязательный 21

Б

База данных: реляционная 349
Базовая линия 108
Библиотека 327
Блок 31, 215
◊ позиций 157
Блочная цитата 29

В

Валидатор 489
Ветвление 215
◊ множественное 216
Вложенность тегов 21
Внедрение зависимостей 413
Всплывающее сообщение 449
Вход 325
Выделение 142
Выражение 203
◊ блочное 215
Выход 325

Г

Гибкая верстка 155
Гиперссылка 56
◊ активная 57
◊ графическая 61
◊ посещенная 57
◊ почтовая 59
◊ пустая 57
◊ текстовая 56
◊ цель 57
Глубина перспективы 177
Горизонтальная линия 33
Горячая клавиша 59
Горячая область 61
Гость 325
Градиент 128
◊ линейный 128
◊ повторяющийся 132
◊ радиальный 128
Группа 78
◊ пунктов списка 74
◊ колонок 51, 52

Д

Дамп 732
Действие 328
Декремент 208
Деструктор 341
Директива 114
Диск 499
Доменное имя 16

З

Заголовок 25
◊ уровень 25
Запись 349
Запрос 353
◊ к базе данных 427
◊ формы 495

Зарегистрированный пользователь 324

И

Идентификатор
◊ интервала 294
◊ тайм-аута 295
Изображение
◊ внешнее 700
◊ гиперссылка 61
◊ карта 61
◊ фоновое 124
Импорт 733
Импортирование 346
Имя
◊ индекса 352
◊ карты 62
◊ объекта 227
◊ переменной 204, 207
◊ полное 345
◊ поля 349
◊ связи 390
◊ таблицы 349
◊ формы 67
◊ функции 221
◊ элемента Web-страницы 64
Индекс 224, 351
◊ атрибут 351
◊ ключевой 351
◊ уникальный 351
Инициализатор 231
Инкремент 208
Интервал 294
Интернет-адрес
◊ абсолютный 58
◊ относительный 59
◊ полный 58
◊ сокращенный 58
◊ физический 417
Интерфейс 342
Исключение 234

К

Канва 687
Карта 61
Каскадная таблица стилей 83
Класс 336
◊ объявление 337
◊ потомок 338
◊ родитель 338
Клиент 15
Ключ 335, 351
◊ внешний 352
Ключевое слово 206
Кнопка 77
◊ графическая 77
◊ отправки данных 77
◊ очистки 77
Коллекция 242
Колонка 52
Комментарий 38, 91, 236
Константа 203
◊ именованная 331
◊ класса 342
Конструктор 340
Контейнер
◊ блочный 31
◊ встроенный 35
Контекст
◊ данных 445
◊ рисования 687
Контроллер 328
◊ действие 452
◊ функция 451
Конфликт стилей 88
Корневая папка 17
Кэширование
◊ на стороне клиента 530
◊ на стороне сервера 530

Л

Лайтбокс 573
Литерал 36, 301
Локальный хост 358

М

Маркер 27
◊ графический 111
Маршрут 405
◊ группа 416
◊ именованный 409
◊ массовое создание 410
◊ параметризованный 407
Маршрутизатор 328
Маска 154, 706
Массив 224
◊ ассоциативный 335
◊ вложенный 225
◊ комбинированный 335
◊ размер 224
◊ элемент 224
Медиазапрос 194
◊ CSS 199
◊ HTML 195
Метаданные 23
Метатег 23
Метод 227
◊ GET 65
◊ POST 66
◊ добавленный 229
◊ защищенный 341
◊ кодирования данных 66
◊ отправки данных 65
◊ переопределение 339
◊ публичный 341
◊ статический 229
◊ частный 341
Миграция 328
Модель 327
◊ внедрение 413
◊ промежуточная 395
Модификатор доступа 341

Н

Наблюдатель 400
Набор состояний 189
Надпись 78
Наследование 338
◊ стилей 90
Неразрывный пробел 37
Номер записи 350

О

Область редактирования 73
Обработчик события 259
◊ по умолчанию 268
◊ привязка 259
Объединение ячеек 53
Объект 227, 336
◊ CanvasRenderingContect2D 687

Объектная модель
◊ Web-обозревателя 285
◊ документа 240
Операнд 204
Оператор 204
◊ арифметический 204, 208
◊ ветвления 217
◊ возврата 221
◊ вывода 331
◊ логический 210
◊ объединения строк 209
◊ объявления глобальных переменных 333
◊ объявления переменной 204, 207
◊ объявления статической переменной 334
◊ перезапуска 220
◊ получения типа 211
◊ прерывания 220
◊ присваивания 204
◊ простого присваивания 209
◊ сложного присваивания 209
◊ создания экземпляра 227
◊ сравнения 210
◊ сравнения экземпляра объекта 232
◊ строгого сравнения 213
Оповещение 522
Откат 372
Отступ
◊ внешний 133
◊ внутренний 133

П

Пагинатор 440
Панель навигации 63
Папка проекта 365
Параметр 221
◊ необязательный 222
Переключатель 72
Переключение 217
Переменная 204
◊ глобальная 222
◊ локальная 222
◊ объявление 204, 207
◊ статическая 334
Переполнение 140
Перо 690
Перспектива 177
Поведение 14
Поле 349
◊ атрибут 350
◊ беззнаковое 357
◊ вычисляемое 397
◊ индексированное 351
◊ ключевое 351
◊ обязательное 350
◊ счетчика 357
◊ тип 350
Поле ввода 69
◊ адреса электронной почты 71
◊ интернет-адреса 71
◊ пароля 70
◊ файла 75
◊ числа 70
Политика 510
Полоса навигации 63
Пользователь 324
Порядок обхода 60
◊ номер 60
Посредник 404
Права пользователя 325
Правила каскадности 90
Представление 14
Преобразование 174
◊ двухмерное 174
◊ сложное 183
◊ трехмерное 174
Приоритет операторов 205, 213
Провайдер 399
Проект 364
Пространство имен 344
Протокол 16
Прототипирование 327
Псевдокласс 96
Псевдоэлемент 100
Пункт списка 27

Р

Разграничение доступа 324
Разделитель 92
Разделяемые данные 470
Разметка 167
◊ рамочная 171
◊ с фиксированными шапкой и поддоном 169
◊ табличная 168
Разрыв строки 35
Раскрывающаяся панель 568
Регулярное выражение 301
◊ группа 303
Регулятор 72
Результат 221

С

Сброс пароля 368
Свойство 227
◊ loop 281
◊ добавленное 229
◊ защищенное 341
◊ публичное 341
◊ статическое 229
◊ частное 341
Связь 352
◊ многие-ко-многим 353
◊ один-к-одному 353
◊ один-ко-многим 352
◊ сквозная 395
Секция Web-страницы 22
◊ заголовка 22
◊ тела 22
Секция таблицы 50
◊ поддона 50
◊ тела 50
◊ шапки 50
Селектор 84
Семантическая разметка 31
Семейство шрифтов 105
Сервер 15
◊ данных 358
Серверная программа 323
◊ стартовая 366
Символ специальный 36, 205
Скрипт 203
Скрытое поле 76
Слаг 614
Событие 258
◊ источник 259
Содержание 14
◊ генерируемое 119
Составитель 471

Список 27, 73
◊ вложенный 28
◊ истории 290
◊ маркированный 27
◊ нумерованный 27
◊ определений 29
◊ раскрывающийся 73
Спойлер 568
Ссылка 226
◊ пустая 226
Стек 469
Стилевой класс 94
Стиль 83
◊ встроенный 87
Строка 46, 205
Структура Web-сайта
◊ логическая 542
◊ физическая 542
СУБД 348
◊ серверная 358
Сценарий 203
Счетчик 120
◊ сброс 121
◊ цикла 218

Т

Таблица 46
◊ вторичная 352
◊ заголовок 49
◊ первичная 352
◊ связующая 353
Таблица стилей 83, 86
◊ внешняя 86
◊ внутренняя 87
Тайм-аут 295
Таймер 294
◊ неповторяющийся 295
◊ повторяющийся 294
Тег 20
◊ дочерний 21
◊ закрывающий 20
◊ имя 20
◊ невидимый 22
◊ одинарный 20
◊ открывающий 20
◊ парный 20
◊ потомок 21
◊ родитель 21
◊ родительский 21
◊ содержимое 20
Текст замены 42
Тип данных 205
◊ undefined 206
◊ логический 206
◊ объектный 225
◊ строковый 205
◊ функциональный 224
◊ числовой 206
Точка
◊ зрения 179
◊ ключевая 128
◊ конечная 128
◊ начальная 128
Трейт 343

Узел 244
Указатель 92
◊ на атрибут тега 95
◊ на параметр устройства 196
◊ на тип устройства 196
◊ основной 93
◊ универсальный 95
Управляющая конструкция 215
Уровень вложенности тега 21
Условие 215

Фаза
◊ всплытия 266
◊ источника 266
◊ туннелирование 266
Файл
◊ Web-сценария 237
◊ конфигурации 328
◊ по умолчанию 17
◊ целевой 58
Фасад 374
Фиксированный формат 30
Флажок 71
Фокус ввода 60
Фон
◊ градиентный 128
◊ графический 124
◊ сплошной 123
Форма 65
Фрейм 79
Фреймворк 326
Функция 221
◊ агрегатная 354
◊ анонимная 224
◊ вызов 222
◊ именованная 224
◊ объявление 221
◊ тело 221

Хостинг-провайдер 14
Хранилище 499, 707
◊ локальное 707
◊ сессионное 707
Хэш 335

Ц

Цвет
◊ графический 699
◊ имя 103
Цикл 218
◊ перезапуск 220
◊ по массиву 332
◊ по свойствам объекта 233
◊ прерывание 220
◊ с постусловием 220
◊ с предусловием 219
◊ со счетчиком 218
◊ тело 218

Ч

Число 206

Ш

Шаблон 327
◊ вложенный 465
◊ наследование 467
◊ потомок 468
◊ родитель 467
◊ секция 467
Шлюз 509
Шрифт загружаемый 113

Э

Экземпляр объекта 227
Экспорт 732
Элемент Web-страницы
◊ блочный 27
◊ внедренный 39
◊ встраивающийся 117
◊ встроенный 35
◊ дочерний 21
◊ непозиционируемый 151
◊ относительно позиционируемый 152
◊ плавающий 149
◊ позиционируемый 152
◊ потомок 21
◊ родитель 21
◊ родительский 21
◊ с прокруткой 141
◊ свободно позиционируемый 152
◊ фиксированный 152
Элемент
◊ оформления 78
◊ управления 65

Я

Якорь 64
Ячейка 47
◊ шапки 47